

Symbolic algebra and Mathematics with Xcas

Renée De Graeve, Bernard Parisse¹,
Jay Belanger²
Sections written by Luka Marohnić³

¹Université de Grenoble, initial translation of parts of the French user manual

²Full translation and improvements

³Optimization, signal processing. The graph theory is in a separate manual.

© 2002, 2007 Renée De Graeve, Bernard Parisse
renee.degraeve@wanadoo.fr
bernard.parisser@ujf-grenoble.fr

Contents

1	Index	33
2	Introduction	51
2.1	Notations used in this manual	51
2.2	Interfaces for the <code>giac</code> library	52
2.2.1	The <code>Xcas</code> interface	53
2.2.2	The command-line interface: <code>icas giac</code>	53
2.2.3	The Firefox interface	54
2.2.4	The TeXmacs interface	54
2.2.5	Checking the version of <code>giac</code> that you are using: <code>version giac</code>	55
3	The <code>Xcas</code> interface	57
3.1	The entry levels	57
3.2	The starting window	58
3.3	Getting help	60
3.4	The menus	63
3.4.1	The <code>File</code> menu	63
3.4.2	The <code>Edit</code> menu	64
3.4.3	The <code>Cfg</code> menu	65
3.4.4	The <code>Help</code> menu	66
3.4.5	The <code>Toolbox</code> menu	69
3.4.6	The <code>Expression</code> menu	69
3.4.7	The <code>Cmds</code> menu	69
3.4.8	The <code>Prg</code> menu	69
3.4.9	The <code>Graphic</code> menu	70
3.4.10	The <code>Geo</code> menu	70
3.4.11	The <code>Spreadsheet</code> menu	70
3.4.12	The <code>Phys</code> menu	70
3.4.13	The <code>Highschool</code> menu	70
3.4.14	The <code>Turtle</code> menu	70
3.5	Configuring <code>Xcas</code>	70
3.5.1	The number of significant digits: <code>Digits DIGITS</code>	70
3.5.2	The language mode: <code>xcas_mode</code>	71
3.5.3	The units for angles: <code>angle_radian</code>	71
3.5.4	Exact or approximate values: <code>approx_mode</code>	72
3.5.5	Complex numbers: <code>cfactor complex_mode</code>	72
3.5.6	Complex variables: <code>complex_variables</code>	73

3.5.7	Configuring the computations	73
3.5.8	Configuring the graphics	76
3.5.9	More configuration	77
3.5.10	The configuration file: <code>widget_size cas_setup xcas_mode xyztrange</code>	78
3.6	Printing and saving	80
3.6.1	Saving a session	80
3.6.2	Saving a spreadsheet	81
3.6.3	Saving a program	81
3.6.4	Printing a session	81
3.7	Translating to other computer languages	82
3.7.1	Translating an expression to L ^A T _E X: <code>latex</code>	82
3.7.2	Translating the entire session to L ^A T _E X	82
3.7.3	Translating graphical output to L ^A T _E X: <code>graph2tex graph3d2tex</code>	82
3.7.4	Translating an expression to MathML: <code>mathml</code>	83
3.7.5	Translating a spreadsheet to MathML	83
3.7.6	Indent an XML string: <code>xml_print</code>	84
3.7.7	Export to presentation or content MathML: <code>export_mathml</code>	84
3.7.8	Translating a Maple file to Xcas: <code>maple2xcas</code>	86
4	Entry in Xcas	87
4.1	Suppressing output: <code>nodisp ;</code>	87
4.2	Entering comments: <code>comment</code>	88
4.3	Editing expressions	88
4.3.1	Entering expressions in the editor: an example	88
4.3.2	Subexpressions	89
4.3.3	Manipulating subexpressions	91
4.4	Previous results: <code>ans</code>	92
4.5	Spreadsheet	92
4.5.1	Opening a spreadsheet	92
4.5.2	The spreadsheet window	93
5	CAS building blocks	95
5.1	Numbers	95
5.2	Symbolic constants: <code>e pi infinity inf i euler_gamma</code>	96
5.3	Sequences, sets and lists	97
5.3.1	Sequences: <code>seq[] ()</code>	97
5.3.2	Sets: <code>set[]</code>	98
5.3.3	Lists: <code>[]</code>	98
5.3.4	Accessing elements	99
5.4	Variables	100
5.4.1	Variable names	100
5.4.2	Assigning values: <code>:= => = assign sto Store</code>	100
5.4.3	Assignment by reference: <code>=<</code>	101
5.4.4	Copying lists: <code>copy</code>	101
5.4.5	Incrementing variables: <code>+= -= *= /=</code>	102

5.4.6	Storing and recalling variables and their values: archive unarchive	103
5.4.7	Copying variables: CopyVar	103
5.4.8	Assumptions on variables: about additionally assume purge supposons and or	104
5.4.9	Unassigning variables: VARs purge DelVar del restart rm_a_z rm_all_vars	107
5.4.10	The CST variable	108
5.5	Functions	109
5.5.1	Defining functions	109
5.6	Directories	110
5.6.1	Working directories: pwd cd	110
5.6.2	Reading files: read load	111
5.6.3	Internal directories: NewFold SetFold GetFold DelFold VARs	112
6	The CAS functions	113
6.1	Booleans	113
6.1.1	Boolean values: true false	113
6.1.2	Tests: == != > >= < <=	113
6.1.3	Defining functions with boolean tests: ifte ?: when	114
6.1.4	Boolean operators: or xor and not	116
6.1.5	Transforming a boolean expression to a list: exp2list	117
6.1.6	Transforming a list into a boolean expression: list2exp	118
6.1.7	Evaluating booleans: evalb	119
6.2	Bitwise operators	119
6.2.1	Basic operators: bitor bitxor bitand	119
6.2.2	Bitwise Hamming distance: hamdist	121
6.3	Strings	121
6.3.1	Characters and strings: "	121
6.3.2	The newline character: \n	122
6.3.3	The length of a string: size length	123
6.3.4	The left and right parts of a string: left right	123
6.3.5	First character, middle and end of a string: head mid tail	124
6.3.6	Concatenation of a sequence of words: cumSum	125
6.3.7	ASCII code of a character: ord	125
6.3.8	ASCII code of a string: asc	126
6.3.9	String defined by the ASCII codes of its characters: char	126
6.3.10	Finding a character in a string: inString	127
6.3.11	Concatenating objects into a string: cat	127
6.3.12	Adding an object to a string: +	128
6.3.13	Transforming a real number into a string: cat +	129
6.3.14	Transforming a string into a number: expr	129
6.4	Writing an integer in a different base	130
6.4.1	Writing an integer in base 2, 8 or 16	130

6.4.2	Writing an integer in an arbitrary base b : convert	132
6.5	Integers (and Gaussian Integers)	133
6.5.1	GCD: gcd igcd Gcd	133
6.5.2	GCD of a list of integers: lgcd	136
6.5.3	The least common multiple: lcm	136
6.5.4	Decomposition into prime factors: ifactor	137
6.5.5	List of prime factors: ifactors	138
6.5.6	Matrix of factors: maple_ifactors	139
6.5.7	The divisors of a number: idivis divisors	140
6.5.8	The integer Euclidean quotient: iquo intDiv div	140
6.5.9	The integer Euclidean remainder: irem remain smod mods mod %	141
6.5.10	Euclidean quotient and Euclidean remainder of two integers: iquorem	143
6.5.11	Test of evenness: even	143
6.5.12	Test of oddness: odd	144
6.5.13	Test of pseudo-primality: is_pseudoprime	144
6.5.14	Test of primality: is_prime isprime isPrime	145
6.5.15	The smallest pseudo-prime greater than n : nextprime	147
6.5.16	The greatest pseudo-prime less than n : prevprime	147
6.5.17	The n th pseudo-prime number: ithprime	147
6.5.18	The number of pseudo-primes less than or equal to n : nprimes	148
6.5.19	Bézout's Identity: iegcd igcdex	148
6.5.20	Solving $au + bv = c$ in \mathbb{Z} : iabcuv	149
6.5.21	Chinese remainders: ichinrem ichrem chrem	149
6.5.22	Solving $a^2 + b^2 = p$ in \mathbb{Z} : pa2b2	152
6.5.23	Solving Diophantine equations: isolve	152
6.5.24	The Euler indicatrix: euler phi	155
6.5.25	Legendre symbol: legendre_symbol	156
6.5.26	Jacobi symbol: jacobi_symbol	157
6.5.27	Listing all compositions of an integer into k parts: icomp	157
6.6	Combinatorial analysis	158
6.6.1	Factorial: factorial !	158
6.6.2	Binomial coefficients: binomial comb nCr	159
6.6.3	Permutations: perm nPr	160
6.6.4	Wilf-Zeilberger pairs: wz_certificate	160
6.7	Rational numbers	162
6.7.1	Transform a floating point number into a rational: exact float2rational	162
6.7.2	Integer and fractional part: propfrac propFrac	163
6.7.3	Numerator of a fraction after simplification: numer getNum	164
6.7.4	Denominator of a fraction after simplification: denom getDenom	164
6.7.5	Numerator and denominator of a fraction: f2nd fxnd	165
6.7.6	Simplifying a pair of integers: simp2	165

6.7.7	Continued fraction representation of a real: dfc	166
6.7.8	Transforming a continued fraction representation into a real: dfc2f	169
6.7.9	The n -th Bernoulli number: bernoulli	170
6.7.10	Accessing to PARI/GP commands: pari	171
6.8	Real numbers	171
6.8.1	Evaluating a real at a given precision: evalf Digits DIGITS	171
6.8.2	The standard infix operators on real numbers: + - * / ^	173
6.8.3	Prefixed division on reals: rdiv	174
6.8.4	n -th root: root	175
6.8.5	The exponential integral function: Ei	175
6.8.6	The logarithmic integral function: Li	177
6.8.7	The cosine integral function: Ci	178
6.8.8	The sine integral function: Si	178
6.8.9	The Heaviside function: Heaviside	179
6.8.10	The Dirac distribution: Dirac	180
6.8.11	Error function: erf	181
6.8.12	Complementary error function: erfc	182
6.8.13	The Γ function: Gamma	183
6.8.14	The upper incomplete γ function: ugamma	184
6.8.15	The lower incomplete γ function: igamma	184
6.8.16	The β function: Beta	185
6.8.17	Derivatives of the DiGamma function: Psi	186
6.8.18	The ζ function: Zeta	187
6.8.19	Airy functions: Airy_Ai and Airy_Bi	187
6.9	Permutations	189
6.9.1	Random permutation: randperm shuffle	189
6.9.2	Previous and next permutation: prevperm nextperm	189
6.9.3	Decomposing a permutation into a product of disjoint cycles: permu2cycles	190
6.9.4	Product of cycles to permutation: cycles2permu	191
6.9.5	Transforming a cycle into a permutation: cycle2perm	192
6.9.6	Transforming a permutation into a matrix: permu2mat	192
6.9.7	Checking for a permutation: is_permu	192
6.9.8	Checking for a cycle: is_cycle	193
6.9.9	Product of two permutations: p1op2 c1op2 p1oc2 c1oc2	194
6.9.10	Signature of a permutation: signature	195
6.9.11	Inverse of a permutation: perminv	195
6.9.12	Inverse of a cycle: cycleinv	196
6.9.13	Order of a permutation: permuorder	196
6.9.14	The group generated by two permutations: groupermu	197
6.10	Complex numbers	197
6.10.1	The usual complex operators: + - * / ^	197
6.10.2	The real and imaginary parts of a complex number: re real im imag	197

6.10.3	Writing a complex number z in rectangular form: evalc	198
6.10.4	The modulus and argument of a complex number: abs arg	199
6.10.5	The normalized complex number: normalize unitV	200
6.10.6	Conjugate of a complex number: conj	200
6.10.7	Multiplication by the complex conjugate: mult_c_conjugate	200
6.10.8	Barycenter of complex numbers: barycenter	201
6.11	Algebraic numbers	202
6.11.1	Definition	202
6.11.2	Minimum polynomial of an algebraic number: pmin	202
6.12	Algebraic expressions	203
6.12.1	Evaluating an expression: eval	203
6.12.2	Changing the evaluation level: eval_level	204
6.12.3	Evaluating algebraic expressions: evala	205
6.12.4	Preventing evaluation: quote hold '	205
6.12.5	Forcing evaluation: unquote	206
6.12.6	Distribution: expand fdistrib	206
6.12.7	Canonical form: canonical_form	206
6.12.8	Multiplication by the conjugate quantity: mult_conjugate	207
6.12.9	Separation of variables: split	208
6.12.10	Factoring: factor cfactor	209
6.12.11	Zeros of an expression: zeros	210
6.12.12	Regrouping expressions: regroup	212
6.12.13	Normal form: normal	212
6.12.14	Simplifying: simplify	213
6.12.15	Automatic simplification: autosimplify	213
6.12.16	Normal form for rational functions: ratnormal	215
6.12.17	Substituting a variable by a value: 	215
6.12.18	Substituting a variable by a value: subst	216
6.12.19	Substituting a variable by a value: ()	218
6.12.20	Substituting a variable by a value (Maple and Mupad compatibility): subs	218
6.12.21	Substituting a subexpression by another expression: algsubs	220
6.12.22	Eliminating one or more variables from a list of equations: eliminate	221
6.12.23	Evaluating a primitive at boundaries: preval	222
6.12.24	Sub-expression of an expression: part	222
6.13	Values of a sequence u_n	223
6.13.1	Array of values of a sequence : tablefunc	223
6.13.2	Values of a recurrence relation or a system: seqsolve	224
6.13.3	Values of a recurrence relation or a system: rsolve	225
6.13.4	Table of values and graph of a recurrent sequence: tableseq	227
6.14	Operators or infix functions	229

6.14.1	Xcas operators: <code>\$ %</code>	229
6.14.2	Defining an operator: <code>user_operator</code>	232
6.15	Functions and expressions with symbolic variables	234
6.15.1	The difference between a function and an expression	234
6.15.2	Transforming an expression into a function: <code>unapply</code>	235
6.15.3	Top and leaves of an expression: <code>sommet feuille op left right</code>	236
6.16	Functions	239
6.16.1	Context-dependent functions.	239
6.16.2	Standard functions	246
6.16.3	Defining algebraic functions	258
6.16.4	Composing functions: <code>@</code>	260
6.16.5	Repeated function composition: <code>@@</code>	261
6.16.6	Defining a function with history: <code>as_function_of</code>	261
6.17	Getting information about functions from \mathbb{R} to \mathbb{R}	262
6.17.1	The domain of a function: <code>domain</code>	262
6.17.2	Table of variations of a function: <code>tabvar</code>	263
6.18	Limits: <code>limit</code>	265
6.19	Derivation and applications	267
6.19.1	Functional derivative: <code>function_diff</code>	267
6.19.2	Length of an arc: <code>arcLen</code>	268
6.19.3	Maximum and minimum of an expression: <code>fMax fMin</code>	269
6.19.4	Derivatives and partial derivatives	271
6.19.5	Implicit differentiation: <code>implicitdiff</code>	274
6.19.6	Numerical differentiation: <code>numdiff</code>	276
6.20	Integration	278
6.20.1	Antiderivative and definite integral: <code>integrate int Int</code>	278
6.20.2	Primitive and definite integral: <code>risch</code>	281
6.20.3	Discrete summation: <code>sum</code>	282
6.20.4	Riemann sum: <code>sum_riemann</code>	284
6.20.5	Integration by parts	286
6.20.6	Change of variables: <code>subst</code>	289
6.20.7	Integrals and limits	289
6.21	Multivariate calculus	290
6.21.1	Gradient: <code>derive deriver diff grad</code>	290
6.21.2	Laplacian: <code>laplacian</code>	291
6.21.3	Hessian matrix: <code>hessian</code>	293
6.21.4	Divergence: <code>divergence</code>	294
6.21.5	Rotational: <code>curl</code>	294
6.21.6	Potential: <code>potential</code>	295
6.21.7	Conservative flux field: <code>vpotential</code>	295
6.21.8	Determining where a function is convex: <code>convex</code>	296
6.22	Calculus of variations	298
6.22.1	The Brachistochrone Problem	298
6.22.2	Euler-Lagrange equation(s): <code>euler_lagrange</code>	299
6.22.3	Solution of the Brachistochrone Problem	303
6.22.4	Jacobi equation: <code>jacobi_equation</code>	304

6.22.5	Finding conjugate points: <code>conjugate_equation</code>	305
6.22.6	An example: Finding the surface of revolution with minimal area	307
6.23	Trigonometry	310
6.23.1	Expanding a trigonometric expression: <code>trigexpand</code>	310
6.23.2	Linearizing a trigonometric expression: <code>tlin</code>	310
6.23.3	Increasing the phase by $\pi/2$ in a trigonometric expression: <code>shift_phase</code>	311
6.23.4	Putting together sine and cosine of the same angle: <code>tcollect</code> <code>tCollect</code>	312
6.23.5	Simplifying: <code>simplify</code>	313
6.23.6	Simplifying trigonometric expressions: <code>trigsimplify</code>	313
6.23.7	Transforming arccos into arcsin: <code>acos2asin</code>	314
6.23.8	Transforming arccos into arctan: <code>acos2atan</code>	314
6.23.9	Transforming arcsin into arccos: <code>asin2acos</code>	314
6.23.10	Transforming arcsin into arctan: <code>asin2atan</code>	315
6.23.11	Transforming arctan into arcsin: <code>atan2asin</code>	315
6.23.12	Transforming arctan into arccos: <code>atan2acos</code>	316
6.23.13	Transforming complex exponentials into sin and cos: <code>sincos</code> <code>exp2trig</code>	316
6.23.14	Transforming $\tan(x)$ into $\sin(x)/\cos(x)$: <code>tan2sincos</code>	317
6.23.15	Transforming $\sin(x)$ into $\cos(x)*\tan(x)$: <code>sin2costan</code>	317
6.23.16	Transforming $\cos(x)$ into $\sin(x)/\tan(x)$: <code>cos2sintan</code>	318
6.23.17	Rewriting $\tan(x)$ in terms of $\sin(2x)$ and $\cos(2x)$: <code>tan2sincos2</code>	318
6.23.18	Rewriting $\tan(x)$ in terms of $\cos(2x)$ and $\sin(2x)$: <code>tan2cossin2</code>	318
6.23.19	Rewriting \sin , \cos , \tan in terms of $\tan(x/2)$: <code>halftan</code>	319
6.23.20	Rewriting trigonometric functions in terms of $\tan(x/2)$ and hyperbolic functions in terms of $\exp(x)$: <code>halftan_hyp2exp</code>	320
6.23.21	Transforming trigonometric functions into complex exponentials: <code>trig2exp</code>	320
6.23.22	Transforming inverse trigonometric functions into logarithms: <code>atrig2ln</code>	321
6.23.23	Simplifying and expressing preferentially with sines: <code>trigsin</code>	321
6.23.24	Simplifying and expressing preferentially with cosines: <code>trigcos</code>	322
6.23.25	Simplifying and expressing preferentially with tangents: <code>trigtan</code>	322
6.23.26	Rewriting an expression with different options: <code>convert</code> <code>convertir =></code>	323
6.24	Exponentials and Logarithms	325
6.24.1	Rewriting hyperbolic functions as exponentials: <code>hyp2exp</code>	325
6.24.2	Expanding exponentials: <code>expexpand</code>	325
6.24.3	Expanding logarithms: <code>lnexpand</code>	326

6.24.4	Linearizing exponentials: <code>lin</code>	326
6.24.5	Collecting logarithms: <code>lncollect</code>	327
6.24.6	Expanding powers: <code>powexpand</code>	327
6.24.7	Rewriting a power as an exponential: <code>pow2exp</code>	328
6.24.8	Rewriting $\exp(n \cdot \ln(x))$ as a power: <code>exp2pow</code>	328
6.24.9	Simplifying complex exponentials: <code>tsimplify</code>	329
6.25	Rewriting transcendental and trigonometric expressions	329
6.25.1	Expanding transcendental and trigonometric expressions: <code>texpand</code> <code>tExpand</code>	329
6.25.2	Combining terms of the same type: <code>combine</code>	331
6.26	Fourier transformation	332
6.26.1	Fourier coefficients: <code>fourier_an</code> and <code>fourier_bn</code> or <code>fourier_cn</code>	332
6.26.2	Continuous Fourier Transform: <code>fourierifourier</code> <code>addtable</code>	336
6.26.3	Discrete Fourier Transform and the Fast Fourier Transform	343
6.26.4	An exercise with <code>fft</code>	349
6.27	Polynomials	351
6.27.1	Polynomials of a single variable: <code>poly1</code>	351
6.27.2	Polynomials of several variables: <code>%%{ %%}</code>	351
6.27.3	Apply a function to the internal sparse format of a polynomial: <code>map</code>	351
6.27.4	Converting to a symbolic polynomial: <code>r2e</code> <code>poly2symp</code>	352
6.27.5	Converting from a symbolic polynomial: <code>e2r</code> <code>symp2poly</code>	353
6.27.6	Transforming a polynomial in internal format into a list, and conversely: <code>convert</code>	354
6.27.7	Coefficients of a polynomial: <code>coeff</code> <code>coeffs</code>	355
6.27.8	Polynomial degree: <code>degree</code>	356
6.27.9	Polynomial valuation: <code>valuation</code> <code>ldegree</code>	357
6.27.10	Leading coefficient of a polynomial: <code>lcoeff</code>	357
6.27.11	Trailing coefficient degree of a polynomial: <code>tcoeff</code>	358
6.27.12	Evaluating polynomials: <code>peval</code> <code>polyEval</code>	359
6.27.13	Factoring x^n in a polynomial: <code>factor_xn</code>	359
6.27.14	GCD of the coefficients of a polynomial: <code>content</code>	360
6.27.15	Primitive part of a polynomial: <code>primpart</code>	360
6.27.16	Factoring: <code>collect</code>	361
6.27.17	Square-free factorization: <code>sqrfree</code>	362
6.27.18	List of factors: <code>factors</code>	363
6.27.19	Evaluating a polynomial: <code>horner</code>	363
6.27.20	Rewriting in terms of the powers of $(x-a)$: <code>ptayl</code>	364
6.27.21	Computing with the exact root of a polynomial: <code>rootof</code>	365
6.27.22	Exact roots of a polynomial: <code>roots</code>	365
6.27.23	Coefficients of a polynomial defined by its roots: <code>pcoeff</code> <code>pcoef</code>	366
6.27.24	Truncating to order n : <code>truncate</code>	367

6.27.25	Converting a series expansion into a polynomial: convert convertir	367
6.27.26	Random polynomial: randpoly randPoly	368
6.27.27	Changing the order of variables: reorder	369
6.27.28	Random lists: ranm	369
6.27.29	Lagrange polynomial: lagrange interp	369
6.27.30	Trigonometric interpolation: triginterp	371
6.27.31	Natural splines: spline	372
6.27.32	Natural interpolation: spline	374
6.27.33	Rational interpolation: thiele	375
6.27.34	Rational interpolation without poles: ratinterp	377
6.28	Arithmetic and polynomials	377
6.28.1	The divisors of a polynomial: divis	378
6.28.2	Euclidean quotient: quo Quo	378
6.28.3	Euclidean remainder: rem Rem	380
6.28.4	Quotient and remainder: quoem divide	381
6.28.5	GCD of two polynomials with the Euclidean algorithm: gcd Gcd	381
6.28.6	GCD of two polynomials with the Euclidean algorithm: Gcd	382
6.28.7	Choosing the GCD algorithm of two polynomials: ezgcd heugcd modgcd psrgcd	383
6.28.8	LCM of two polynomials: lcm	384
6.28.9	Bézout's Identity: egcd gcdex	385
6.28.10	Solving $au + bv = c$ over polynomials: abcuv	386
6.28.11	Chinese remainders: chinrem	387
6.28.12	Cyclotomic polynomial: cyclotomic	388
6.28.13	Sturm sequences and number of sign changes of P on $(a, b]$: sturm sturmseq sturmab	389
6.28.14	Sylvester matrix of two polynomials and resultant: sylvester resultant	393
6.29	Exact bounds for roots of a polynomial	397
6.29.1	Exact bounds for real roots of a polynomial: realroot	397
6.29.2	Exact bounds for complex roots of a polynomial: complexroot	398
6.29.3	Exact bounds for real roots of a polynomial: VAS	399
6.29.4	Exact bounds for positive real roots of a polynomial: VAS_positive	400
6.29.5	An upper bound for the positive real roots of a poly- nomial: posubLMQ	400
6.29.6	A lower bound for the positive real roots of a poly- nomial: poslbdLMQ	401
6.29.7	Exact values of rational roots of a polynomial: rationalroot	402
6.29.8	Exact values of the complex rational roots of a poly- nomial: crationalroot	403
6.30	Orthogonal polynomials	403
6.30.1	Legendre polynomials: legendre	403

6.30.2	Hermite polynomial: hermite	404
6.30.3	Laguerre polynomials: laguerre	405
6.30.4	Tchebychev polynomials of the first kind: tchebyshev1	406
6.30.5	Tchebychev polynomial of the second kind: tchebyshev2	407
6.31	Gröbner basis and Gröbner reduction	408
6.31.1	Gröbner basis: gbasis	408
6.31.2	Gröbner reduction: greduce	409
6.31.3	Testing if a polynomial or list of polynomials belongs to an ideal given by a Gröbner basis: in_ideal	410
6.31.4	Building a polynomial from its evaluation: genpoly	411
6.32	Rational functions	413
6.32.1	Numerator: getNum	413
6.32.2	Numerator after simplification: numer	413
6.32.3	Denominator: getDenom	414
6.32.4	Denominator after simplification: denom	414
6.32.5	Numerator and denominator: f2nd fxnd	415
6.32.6	Simplifying: simp2	415
6.32.7	Common denominator: comDenom	416
6.32.8	Polynomial and fractional part: propfrac	416
6.32.9	Partial fraction expansion: partfrac cpartfrac	416
6.33	Exact roots and poles	417
6.33.1	Roots and poles of a rational function: froot	417
6.33.2	Rational function given by roots and poles: fcoeff	418
6.34	Computing in $\mathbb{Z}/p\mathbb{Z}$ or in $\mathbb{Z}/p\mathbb{Z}[x]$	419
6.34.1	Expanding and reducing: normal	419
6.34.2	Addition in $\mathbb{Z}/p\mathbb{Z}$ or in $\mathbb{Z}/p\mathbb{Z}[x]$: +	420
6.34.3	Subtraction in $\mathbb{Z}/p\mathbb{Z}$ or in $\mathbb{Z}/p\mathbb{Z}[x]$: -	420
6.34.4	Multiplication in $\mathbb{Z}/p\mathbb{Z}$ or in $\mathbb{Z}/p\mathbb{Z}[x]$: *	421
6.34.5	Euclidean quotient : quo	421
6.34.6	Euclidean remainder: rem	422
6.34.7	Euclidean quotient and euclidean remainder: quorem	422
6.34.8	Division in $\mathbb{Z}/p\mathbb{Z}$ or in $\mathbb{Z}/p\mathbb{Z}[x]$: /	423
6.34.9	Power in $\mathbb{Z}/p\mathbb{Z}$ and in $\mathbb{Z}/p\mathbb{Z}[x]$: ^	424
6.34.10	Computing $a^n \bmod p$: powmod powermod	424
6.34.11	Inverse in $\mathbb{Z}/p\mathbb{Z}$: inv inverse /	425
6.34.12	Rebuilding a fraction from its value modulo p : fracmod iratrecon	425
6.34.13	GCD in $\mathbb{Z}/p\mathbb{Z}[x]$: gcd	426
6.34.14	Factoring over $\mathbb{Z}/p\mathbb{Z}[x]$: factor factoriser	427
6.34.15	Determinant of a matrix in $\mathbb{Z}/p\mathbb{Z}$: det	427
6.34.16	Inverse of a matrix with coefficients in $\mathbb{Z}/p\mathbb{Z}$: inv inverse	428
6.34.17	Row reduction to echelon form in $\mathbb{Z}/p\mathbb{Z}$: rref	428
6.34.18	Construction of a Galois field: GF	429
6.34.19	Factoring a polynomial with coefficients in a Galois field: factor	432

6.35	Computing in $\mathbb{Z}/p\mathbb{Z}[x]$ using Maple syntax	433
6.35.1	Euclidean quotient: Quo	433
6.35.2	Euclidean remainder: Rem	434
6.35.3	GCD in $\mathbb{Z}/p\mathbb{Z}[x]$: Gcd	435
6.35.4	Factoring in $\mathbb{Z}/p\mathbb{Z}[x]$: Factor	436
6.35.5	Determinant of a matrix with coefficients in $\mathbb{Z}/p\mathbb{Z}$: Det	436
6.35.6	Inverse of a matrix in $\mathbb{Z}/p\mathbb{Z}$: Inverse	437
6.35.7	Row reduction to echelon form in $\mathbb{Z}/p\mathbb{Z}$: Rref	438
6.36	Taylor and asymptotic expansions	439
6.36.1	Dividing by increasing power order: divpc	439
6.36.2	Series expansion: taylor series	439
6.36.3	The inverse of a series: revert	442
6.36.4	The residue of an expression at a point: residue	443
6.36.5	Padé expansion: pade	443
6.37	Ranges of values	445
6.37.1	Definition of a range of values: ...	445
6.37.2	Center of a range of values: interval2center	446
6.37.3	Ranges of values defined by their center: center2interval	446
6.38	Intervals	447
6.38.1	Defining intervals: i[]	447
6.38.2	The endpoints of an interval: left right	448
6.38.3	Interval arithmetic: + - * /	448
6.38.4	The midpoint of an interval: midpoint	450
6.38.5	The union of intervals: union	450
6.38.6	The intersection of intervals: intersect	451
6.38.7	Testing if an object is in an interval: contains	451
6.38.8	Converting a number into an interval: convert	452
6.38.9	Converting box constraints from matrix to interval form: box_constraints	452
6.39	Sequences and lists	453
6.39.1	Defining a sequence or a list: seq[] ()	453
6.39.2	Making a sequence or a list: seq \$	454
6.39.3	Length of a sequence or list: size nops length	457
6.39.4	Getting the first element of a sequence or list: head	458
6.39.5	Getting a sequence or list without the first element: tail	459
6.39.6	Getting an element of a sequence or a list: [] [[]] at	459
6.39.7	Finding a subsequence or a sublist	460
6.39.8	Concatenating two sequences: ,	461
6.39.9	The + operator applied on sequences and lists	462
6.39.10	Transforming sequences into lists and lists into sequences: [] nop op makesuite	462
6.40	Operations on lists	463
6.40.1	Sizes of a list of lists: sizes	463
6.40.2	Making a list with a function: makelist	464
6.40.3	Making a list with zeros: newList	465
6.40.4	Making a list of integers: range	465

6.40.5	Selecting elements of a list: select	467
6.40.6	The left and right portions of a list: left right	467
6.40.7	Modifying the elements of a list: subsop	468
6.40.8	Removing an element in a list: suppress	469
6.40.9	Removing elements of a list: remove	470
6.40.10	Inserting an element into a list or a string: insert	471
6.40.11	Appending an element at the end of a list: append	471
6.40.12	Prepending an element at the beginning of a list: prepend	472
6.40.13	Concatenating two lists or a list and an element: concat augment	472
6.40.14	Flattening a list: flatten	474
6.40.15	Reversing order in a list: revlist	474
6.40.16	Rotating a list: rotate	475
6.40.17	Shifting the elements of a list: shift	475
6.40.18	Sorting: sort	476
6.40.19	Sorting a list by increasing order: SortA	477
6.40.20	Sorting a list by decreasing order: SortD	478
6.40.21	Number of elements equal to a given value: count_eq	478
6.40.22	Number of elements smaller than a given value: count_inf	479
6.40.23	Number of elements greater than a given value: count_sup	479
6.40.24	Sum of elements of a list: sum add	479
6.40.25	Sum of list (or matrix) elements transformed by a function: count	480
6.40.26	Cumulated sum of the elements of a list: cumSum	481
6.40.27	Product: product mul	482
6.40.28	Applying a function of one variable to the elements of a list: map apply	484
6.40.29	Applying a bivariate function to the elements of two lists: zip	486
6.40.30	Folding operators: foldl foldr	487
6.40.31	List of differences of consecutive terms: deltalist	488
6.40.32	Making a matrix with a list: list2mat	488
6.40.33	Making a list with a matrix: mat2list	489
6.41	Operations on sets and lists	489
6.41.1	Defining a set or list: set[] %{ %}	489
6.41.2	Testing if a value is in a list or a set: member contains	490
6.41.3	Union of two sets or of two lists: union	491
6.41.4	Intersection of two sets or of two lists: intersect	491
6.41.5	Difference of two sets or of two lists: minus	492
6.42	Functions for vectors	493
6.42.1	Norms of a vector: maxnorm l1norm l2norm norm	493
6.42.2	Normalizing a vector: normalize unitV	493
6.42.3	Term by term sum of two lists: + .+	494
6.42.4	Term by term difference of two lists: - .-	495
6.42.5	Term by term product of two lists: .*	495

6.42.6	Term by term quotient of two lists: <code>./</code>	495
6.42.7	Scalar product : <code>scalar_product * dotprod dot dotP</code> <code>scalar_Product</code>	496
6.42.8	Cross product: <code>cross crossP crossproduct</code>	496
6.42.9	Statistics on lists: <code>mean variance stddev stddevp</code> <code>median quantile quartiles boxwhisker</code>	497
6.43	Tables with strings as indices: <code>table</code>	501
6.44	Matrices	502
6.44.1	Matrices	502
6.44.2	Special matrices	502
6.44.3	Combining matrices	505
6.44.4	Creating a matrix with a formula or function: <code>makemat</code> <code>matrix</code>	510
6.44.5	Getting the parts of a matrix	511
6.44.6	Modifying matrices	516
6.45	Arithmetic and matrices	526
6.45.1	Evaluating a matrix: <code>evalm</code>	526
6.45.2	Addition and subtraction of two matrices: <code>+</code> <code>-</code> <code>+</code> <code>-</code>	526
6.45.3	Multiplication of two matrices: <code>*</code> <code>&*</code>	527
6.45.4	Addition of elements of a column of a matrix: <code>sum</code>	528
6.45.5	Cumulated sum of elements of each column of a matrix: <code>cumSum</code>	528
6.45.6	Multiplication of elements of each column of a matrix: <code>product</code>	528
6.45.7	Power of a matrix: <code>^</code> <code>&^</code>	529
6.45.8	Hadamard product: <code>hadamard product .*</code>	529
6.45.9	Hadamard division: <code>./</code>	530
6.45.10	Hadamard power: <code>.^</code>	530
6.45.11	The elementary row operations	530
6.45.12	Counting the elements of a matrix satisfying a property: <code>count</code>	533
6.45.13	Counting the elements equal to a given value: <code>count_eq</code>	534
6.45.14	Counting the elements smaller than a given value: <code>count_inf</code>	535
6.45.15	Counting the elements greater than a given value: <code>count_sup</code>	535
6.45.16	Statistics functions acting on column matrices: <code>mean</code> <code>stddev variance median quantile quartiles boxwhisker</code>	535
6.45.17	Dimension of a matrix: <code>dim</code>	538
6.45.18	Number of rows: <code>rowdim rowDim nrows</code>	538
6.45.19	Number of columns: <code>coldim colDim ncols</code>	539
6.46	Sparse matrices	539
6.46.1	Defining sparse matrices	539
6.46.2	Operations on sparse matrices	540
6.47	Linear algebra	541
6.47.1	Transpose of a matrix: <code>tran transpose</code>	541

6.47.2	Inverse of a matrix: <code>inv /</code>	542
6.47.3	Trace of a matrix: <code>trace</code>	542
6.47.4	Determinant of a matrix: <code>det</code>	542
6.47.5	Determinant of a sparse matrix: <code>det_minor</code>	543
6.47.6	Rank of a matrix: <code>rank</code>	544
6.47.7	Transconjugate of a matrix: <code>trn</code>	544
6.47.8	Equivalent matrix: <code>changebase</code>	544
6.47.9	Basis of a linear subspace : <code>basis</code>	545
6.47.10	Basis of the intersection of two subspaces: <code>ibasis</code>	545
6.47.11	Image of a linear function: <code>image</code>	546
6.47.12	Kernel of a linear function: <code>kernel nullspace ker</code>	546
6.47.13	Kernel of a linear function: <code>Nullspace</code>	547
6.47.14	Subspace generated by the columns of a matrix: <code>colspace</code>	547
6.47.15	Subspace generated by the rows of a matrix: <code>rowspace</code>	548
6.48	Matrix reduction	549
6.48.1	Eigenvalues: <code>eigenvals</code>	549
6.48.2	Eigenvalues: <code>egvl eigenvalues eigVl</code>	550
6.48.3	Eigenvectors: <code>egv eigenvectors eigenvects eigVc</code>	551
6.48.4	Rational Jordan matrix: <code>rat_jordan</code>	551
6.48.5	Jordan normal form: <code>jordan</code>	553
6.48.6	Powers of a square matrix: <code>matpow</code>	554
6.48.7	Characteristic polynomial: <code>charpoly</code>	555
6.48.8	Characteristic polynomial using Hessenberg algorithm: <code>pcar_hessenberg</code>	555
6.48.9	Minimal polynomial: <code>pmin</code>	556
6.48.10	Adjoint matrix: <code>adjoint_matrix</code>	557
6.48.11	Companion matrix of a polynomial: <code>companion</code>	559
6.48.12	Hessenberg matrix reduction: <code>hessenberg SCHUR</code>	559
6.48.13	Hermite normal form: <code>ihermite</code>	561
6.48.14	Smith normal form in \mathbb{Z} : <code>ismith</code>	562
6.48.15	Smith normal form: <code>smith</code>	563
6.49	Matrix factorizations	564
6.49.1	Cholesky decomposition: <code>cholesky</code>	564
6.49.2	QR decomposition: <code>qr</code>	565
6.49.3	QR decomposition (for TI compatibility): <code>QR</code>	566
6.49.4	LQ decomposition (HP compatible): <code>LQ</code>	567
6.49.5	LU decomposition: <code>lu</code>	567
6.49.6	LU decomposition (for TI compatibility): <code>LU</code>	569
6.49.7	Singular values (HP compatible): <code>SVL svl</code>	570
6.49.8	Singular value decomposition: <code>svd</code>	571
6.49.9	Short basis of a lattice: <code>lll</code>	571
6.50	Different matrix norms	573
6.50.1	The Frobenius norm: <code>frobenius_norm</code>	573
6.50.2	ℓ^2 matrix norm: <code>norm l2norm</code>	573
6.50.3	ℓ^∞ matrix norm: <code>maxnorm</code>	574
6.50.4	Matrix row norm: <code>rownorm rowNorm</code>	574
6.50.5	Matrix column norm: <code>colnorm colNorm l1norm</code>	575

6.50.6	The operator norm of a matrix: <code>matrix_norm</code> <code>l1norm</code> <code>l2norm</code> <code>norm</code> <code>specnorm</code> <code>linfnorm</code>	575
6.51	Isometries	577
6.51.1	Recognizing an isometry: <code>isom</code>	578
6.51.2	Finding the matrix of an isometry: <code>mkisom</code>	579
6.52	Linear Programming	581
6.52.1	Simplex algorithm: <code>simplex_reduce</code>	581
6.52.2	Solving general linear programming problems: <code>lpsolve</code>	585
6.52.3	Solving the transportation problems: <code>tpsolve</code>	597
6.53	Nonlinear optimization	599
6.53.1	Global extrema: <code>minimize</code> <code>maximize</code>	599
6.53.2	Local extrema: <code>extrema</code>	602
6.53.3	Continuous derivative-free optimization: <code>nlpsolve</code>	604
6.53.4	Minimax polynomial approximation: <code>minimax</code>	608
6.54	Quadratic forms	609
6.54.1	Matrix of a quadratic form: <code>q2a</code>	609
6.54.2	Transforming a matrix into a quadratic form: <code>a2q</code>	609
6.54.3	Reducing a quadratic form: <code>gauss</code>	610
6.54.4	The conjugate gradient algorithm: <code>conjugate_gradient</code>	610
6.54.5	Gram-Schmidt orthonormalization: <code>gramschmidt</code>	611
6.54.6	Graph of a conic: <code>conic</code>	612
6.54.7	Conic reduction: <code>reduced_conic</code>	612
6.54.8	Graph of a quadric: <code>quadric</code>	614
6.54.9	Quadric reduction: <code>reduced_quadric</code>	614
6.55	Equations	616
6.55.1	Defining an equation: <code>equal</code>	616
6.55.2	Transforming an equation into a difference: <code>equal2diff</code>	617
6.55.3	Transforming an equation into a list: <code>equal2list</code>	617
6.55.4	The left member of an equation: <code>left</code> <code>gauche</code> <code>lhs</code>	618
6.55.5	The right member of an equation: <code>right</code> <code>droit</code> <code>rhs</code>	618
6.55.6	Solving equation(s): <code>solve</code> <code>cSolve</code>	619
6.56	Linear systems	621
6.56.1	Matrix of a system: <code>syst2mat</code>	621
6.56.2	Gauss reduction of a matrix: <code>ref</code>	622
6.56.3	Gauss-Jordan reduction: <code>rref</code> <code>gaussjord</code>	623
6.56.4	Solving $AX = b$: <code>simult</code>	624
6.56.5	Step by step Gauss-Jordan reduction of a matrix: <code>pivot</code>	625
6.56.6	Linear system solving: <code>linsolve</code>	626
6.56.7	Solving a linear system using the Jacobi iteration method: <code>jacobi_linsolve</code>	629
6.56.8	Solving a linear system using the Gauss-Seidel iteration method: <code>gauss_seidel_linsolve</code>	630
6.56.9	The least squares solution of a linear system: <code>LSQ</code> <code>lsq</code>	631
6.56.10	Finding linear recurrences: <code>reverse_rsolve</code>	632

6.57	Differential equations	633
6.57.1	Solving differential equations: <code>desolve deSolve dsolve</code>	633
6.57.2	Laplace transform and inverse Laplace transform: <code>laplace</code> <code>ilaplace invlaplace</code>	641
6.57.3	Solving linear homogeneous second-order ODE with rational coefficients: <code>kovacicssols</code>	643
6.58	The Z-transform	648
6.58.1	The Z-transform of a sequence: <code>ztrans</code>	648
6.58.2	The inverse Z-transform of a rational function: <code>invztrans</code>	649
6.59	Other functions	650
6.59.1	Replacing small values by 0: <code>epsilon2zero</code>	650
6.59.2	List of variables: <code>lname indets</code>	651
6.59.3	List of variables and of expressions: <code>lvar</code>	652
6.59.4	List of variables of an algebraic expressions: <code>algvar</code>	652
6.59.5	Testing if a variable is in an expression: <code>has</code>	653
6.59.6	Numeric evaluation: <code>evalf</code>	654
6.59.7	Rational approximation: <code>float2rational exact</code>	654
6.60	The day of the week: <code>dayofweek</code>	655
7	Metric properties of curves	657
7.1	The center of curvature	657
7.2	Computing the curvature and related values: <code>curvature osculating_circle</code> <code>evolute</code>	657
8	Graphs	661
8.1	Generalities	661
8.2	The graphic screen	662
8.3	Graph and geometric objects attributes	663
8.3.1	Individual attributes	663
8.3.2	Global attributes	665
8.4	Graph of a function: <code>plotfunc funcplot DrawFunc Graph</code>	667
8.4.1	2-d graph	667
8.4.2	3-d graph	668
8.5	2d graph for Maple compatibility: <code>plot</code>	673
8.6	3d surfaces for Maple compatibility <code>plot3d</code>	674
8.7	Graph of a line and tangent to a graph	676
8.7.1	Drawing a line: <code>line</code>	676
8.7.2	Drawing a 2D horizontal line: <code>LineHorz</code>	678
8.7.3	Drawing a 2D vertical line: <code>LineVert</code>	679
8.7.4	Tangent to a 2D graph: <code>LineTan</code>	679
8.7.5	Tangent to a 2D graph: <code>tangent</code>	680
8.7.6	Plotting a line with a point and the slope: <code>DrawSlp</code>	681
8.7.7	Intersection of a 2D graph with the axis	682
8.8	Graphing inequalities with two variables: <code>plotinequation</code> <code>inequationplot</code>	682
8.9	The area under a curve: <code>area</code>	683

8.10	Graphing the area below a curve: <code>plotarea areaplot</code>	685
8.11	Contour lines: <code>plotcontour contourplot DrwCtour</code>	686
8.12	2-d graph of a 2-d function with colors: <code>plotdensity densityplot</code>	688
8.13	Implicit graph: <code>plotimplicit implicitplot</code>	689
8.13.1	2D implicit curve	689
8.13.2	3D implicit surface	691
8.14	Parametric curves and surfaces: <code>plotparam paramplot DrawParm</code>	692
8.14.1	2D parametric curve	692
8.14.2	3D parametric surface: <code>plotparam paramplot DrawParm</code>	693
8.15	Bezier curves: <code>bezier</code>	695
8.16	Defining curves in polar coordinates: <code>plotpolar polarplot DrawPol courbe_polaire</code>	696
8.17	Graphing recurrent sequences: <code>plotseq seqplot graphe_suite</code>	697
8.18	Tangent field: <code>plotfield fieldplot</code>	698
8.19	Plotting a solution of a differential equation: <code>plotode odeplot</code>	699
8.20	Interactive plotting of solutions of a differential equation: <code>interactive_plotode interactive_odeplot</code>	701
8.21	Animated graphs (2D, 3D or "4D")	702
8.21.1	Animation of a 2D graph: <code>animate</code>	702
8.21.2	Animation of a 3D graph: <code>animate3d</code>	703
8.21.3	Animation of a sequence of graphic objects: <code>animation</code>	704
9	Statistics	711
9.1	One variable statistics	711
9.1.1	The mean: <code>mean</code>	711
9.1.2	Variance: <code>variance</code>	712
9.1.3	Standard deviation: <code>stdev</code>	713
9.1.4	The population standard deviation: <code>stddevp stdDev</code>	714
9.1.5	The median: <code>median</code>	715
9.1.6	Quartiles: <code>quartiles quartile1 quartile3</code>	715
9.1.7	Quantiles: <code>quantile</code>	716
9.1.8	The boxwhisker: <code>boxwhisker mustache</code>	717
9.1.9	Classes: <code>classes</code>	718
9.1.10	Histograms: <code>histogram histogramme</code>	719
9.1.11	Accumulating terms: <code>accumulate_head_tail</code>	720
9.1.12	Frequencies: <code>frequencies frequencies</code>	720
9.1.13	Cumulative frequencies: <code>cumulated_frequencies frequencies_cumulees</code>	721
9.1.14	Bar graphs: <code>bar_plot</code>	723
9.1.15	Pie charts: <code>camembert</code>	724
9.2	Two variable statistics	725

9.2.1	Covariance and correlation: <code>covariance</code> <code>correlation</code> <code>covariance_correlation</code>	725
9.2.2	Scatterplots: <code>scatterplot</code> <code>nuaged_points</code> <code>batons</code> . .	727
9.2.3	Polygonal paths: <code>polygonplot</code> <code>ligne_polygonale</code> <code>linear_interpolate</code> <code>listplot</code> <code>plotlist</code>	728
9.2.4	Linear regression: <code>linear_regression</code> <code>linear_regression_plot</code>	731
9.2.5	Exponential regression: <code>exponential_regression</code> <code>exponential_regression_plot</code>	733
9.2.6	Logarithmic regression: <code>logarithmic_regression</code> <code>logarithmic_regression_plot</code>	734
9.2.7	Power regression: <code>power_regression</code> <code>power_regression_plot</code>	735
9.2.8	Polynomial regression: <code>polynomial_regression</code> <code>polynomial_regression_plot</code>	736
9.2.9	Logistic regression: <code>logistic_regression</code> <code>logistic_regression_plot</code>	737
9.3	Random numbers	739
9.3.1	Producing uniformly distributed random numbers: <code>rand</code> <code>random</code> <code>alea</code> <code>hasard</code> <code>sample</code>	739
9.3.2	Initializing the random number generator: <code>srand</code> <code>randseed</code> <code>RandSeed</code>	742
9.3.3	Producing random numbers with the binomial distribution: <code>randbinomial</code>	742
9.3.4	Producing random numbers with a multinomial distribution: <code>randmultinomial</code>	743
9.3.5	Producing random numbers with a Poisson distribution: <code>randpoisson</code>	743
9.3.6	Producing random numbers with a normal distribution: <code>randnorm</code> <code>randNorm</code>	744
9.3.7	Producing random numbers with Student's distribution: <code>randstudent</code>	744
9.3.8	Producing random numbers with the χ^2 distribution: <code>randchisquare</code>	745
9.3.9	Producing random numbers with the Fisher-Snédecor distribution: <code>randfisher</code>	745
9.3.10	Producing random numbers with the gamma distribution: <code>randgammad</code>	745
9.3.11	Producing random numbers with the beta distribution: <code>randbetad</code>	746
9.3.12	Producing random numbers with the geometric distribution: <code>randgeometric</code>	746
9.3.13	Producing random numbers with the exponential distribution: <code>randexp</code>	746
9.3.14	Random variables: <code>random_variable</code> <code>randvar</code>	747
9.3.15	Make a random vector or list: <code>randvector</code>	756
9.3.16	Producing random matrices: <code>randmatrix</code> <code>ranm</code> <code>randMat</code>	757

9.4	Density and distribution functions	759
9.4.1	Distributions and inverse distributions	759
9.4.2	The uniform distribution	759
9.4.3	The binomial distribution	761
9.4.4	The negative binomial distribution	763
9.4.5	The multinomial probability function: <code>multinomial</code>	765
9.4.6	The Poisson distribution	766
9.4.7	Normal distributions	767
9.4.8	Student's distribution	770
9.4.9	The χ^2 distribution	773
9.4.10	The Fisher-Snedécor distribution	775
9.4.11	The gamma distribution	777
9.4.12	The beta distribution	779
9.4.13	The geometric distribution	780
9.4.14	The Cauchy distribution	782
9.4.15	The exponential distribution	784
9.4.16	The Weibull distribution	785
9.4.17	The Kolmogorov-Smirnov distribution: <code>kolmogorovd</code>	788
9.4.18	The Wilcoxon or Mann-Whitney distribution	788
9.4.19	Moment generating functions for probability distributions: <code>mgf</code>	790
9.4.20	Cumulative distribution functions: <code>cdf</code>	791
9.4.21	Inverse distribution functions: <code>icdf</code>	792
9.4.22	Kernel density estimation: <code>kernel_density kde</code>	792
9.4.23	Distribution fitting by maximum likelihood: <code>fitdistr</code>	795
9.4.24	Markov chains: <code>markov</code>	797
9.4.25	Generating a random walks: <code>randmarkov</code>	798
9.5	Hypothesis testing	799
9.5.1	General	799
9.5.2	Testing the mean with the Z test: <code>normalt</code>	799
9.5.3	Testing the mean with the T test: <code>studentt</code>	800
9.5.4	Testing a distribution with the χ^2 distribution: <code>chisquaret</code>	801
9.5.5	Testing a distribution with the Kolmogorov-Smirnov distribution: <code>kolmogorovt</code>	803
10	Numerical computations	805
10.1	Floating point representation.	805
10.1.1	Digits	805
10.1.2	Representation by hardware floats	806
10.1.3	Examples of representations of normalized floats	806
10.1.4	Difference between the representation of (3.1-3) and of 0.1	807
10.2	Approximate evaluation: <code>evalf approx Digits</code>	808
10.3	Numerical algorithms	809
10.3.1	Approximating solution of an equation: <code>newton</code>	809
10.3.2	Approximating computation of the derivative number: <code>nDeriv</code>	810

10.3.3	Approximating computation of integrals: <code>romberg nInt</code>	811
10.3.4	Approximating integrals with an adaptive Gaussian quadrature at 15 points: <code>gaussquad</code>	811
10.3.5	Approximating solutions of $y'=f(t,y)$: <code>odesolve</code>	812
10.3.6	Approximating solutions of the system $v'=f(t,v)$: <code>odesolve</code>	814
10.3.7	Approximating solutions of nonlinear second-order boundary value problems: <code>bvpsolve</code>	815
10.4	Solving equations with <code>fsolve nSolve cfsolve</code>	818
10.4.1	<code>fsolve</code> with the option <code>bisection_solver</code>	819
10.4.2	<code>fsolve</code> with the option <code>brent_solver</code>	819
10.4.3	<code>fsolve</code> with the option <code>falsepos_solver</code>	820
10.4.4	<code>fsolve</code> with the option <code>newton_solver</code>	820
10.4.5	<code>fsolve</code> with the option <code>secant_solver</code>	820
10.4.6	<code>fsolve</code> with the option <code>steffenson_solver</code>	821
10.5	Solving systems with <code>fsolve</code> and <code>cfsolve</code>	821
10.5.1	<code>fsolve</code> with the option <code>dnewton_solver</code>	823
10.5.2	<code>fsolve</code> with the option <code>hybrid_solver</code>	823
10.5.3	<code>fsolve</code> with the option <code>hybrids_solver</code>	823
10.5.4	<code>fsolve</code> with the option <code>newtonj_solver</code>	823
10.5.5	<code>fsolve</code> with the option <code>hybridj_solver</code>	824
10.5.6	<code>fsolve</code> with the option <code>hybridsj_solver</code>	824
10.6	Numeric roots of a polynomial: <code>proot</code>	824
10.7	Numeric factorization of a matrix: <code>cholesky qr lu svd</code>	825
11	Unit objects and physical constants	827
11.1	Unit objects	827
11.1.1	Notation of unit objects	827
11.1.2	Computing with units	829
11.1.3	Converting units into MKSA units: <code>mksa</code>	830
11.1.4	Converting units: <code>convert =></code>	830
11.1.5	Converting between Celsius and Fahrenheit: <code>Celsius2Fahrenheit Fahrenheit2Celsius</code>	831
11.1.6	Factoring a unit: <code>ufactor</code>	832
11.1.7	Simplifying units: <code>usimplify</code>	832
11.2	Constants	833
11.2.1	Notation of physical constants	833
12	Programming	835
12.1	Functions, programs and scripts	835
12.1.1	The program editor	835
12.1.2	Functions: <code>function endfunction { } local return</code>	835
12.1.3	Local variables	836
12.1.4	Default values of the parameters	837
12.1.5	Programs	837
12.1.6	Scripts	838
12.1.7	Code blocks	838

12.2	Basic instructions	838
12.2.1	Comments: <code>//</code>	838
12.2.2	Input: <code>input Input InputStr textinput output Output</code>	838
12.2.3	Reading a single keystroke: <code>getKey</code>	840
12.2.4	Checking conditions: <code>assert</code>	840
12.2.5	Checking the type of the argument: <code>type subtype</code> <code>compare getType</code>	841
12.2.6	Printing: <code>print Disp ClrIO</code>	843
12.2.7	Displaying exponents: <code>printpow</code>	844
12.2.8	Infix assignments: <code>=> := =<</code>	845
12.2.9	Assignment by copying: <code>copy</code>	846
12.2.10	The difference between <code>:=</code> and <code>=<</code>	847
12.3	Control structures	848
12.3.1	if statements: <code>if then else end elif</code>	848
12.3.2	The switch statement: <code>switch case default</code>	850
12.3.3	The for loop: <code>for from to step do end_for</code>	851
12.3.4	The repeat loop: <code>repeat until</code>	853
12.3.5	The while loop: <code>while</code>	853
12.3.6	Breaking out a loop: <code>break</code>	854
12.3.7	Going to the next iteration of a loop: <code>continue</code>	854
12.3.8	Changing the order of execution: <code>goto label</code>	855
12.4	Other useful instructions	855
12.4.1	Defining a function with a variable number of arguments: <code>args</code>	855
12.4.2	Assignments in a program	856
12.4.3	Writing variable values to a file: <code>write</code>	857
12.4.4	Writing output to a file: <code>fopen fclose fprintf</code>	858
12.4.5	Using strings as names: <code>make_symbol</code>	859
12.4.6	Using strings as commands: <code>expr</code>	860
12.4.7	Converting an expression to a string: <code>string</code>	862
12.4.8	Converting a real number into a string: <code>format</code>	862
12.4.9	Working with the graphics screen: <code>DispG DispHome</code> <code>ClrGraph ClrDraw</code>	863
12.4.10	Pausing a program: <code>Pause WAIT</code>	864
12.4.11	Dealing with errors: <code>try catch throw error ERROR</code>	865
12.5	Debugging	867
12.5.1	Starting the debugger: <code>debug sst in sst_in cont</code> <code>kill break breakpoint halt rmbkr rmbreakpoint watch</code> <code>rmwch</code>	867
13	Two-dimensional Graphics	871
13.1	Introduction	871
13.1.1	Points, vectors and complex numbers	871
13.2	Basic commands	872
13.2.1	Clearing the <code>DispG</code> screen: <code>erase</code>	872
13.2.2	Toggling the axes: <code>switch_axes</code>	872

13.2.3	Drawing unit vectors in the plane: <code>0x_2d_unit_vector</code> <code>0y_2d_unit_vector</code> <code>frame_2d</code>	873
13.2.4	Drawing dotted paper: <code>dot_paper</code>	873
13.2.5	Drawing lined paper: <code>line_paper</code>	874
13.2.6	Drawing grid paper: <code>grid_paper</code>	875
13.2.7	Drawing triangular paper: <code>triangle_paper</code>	875
13.3	Display features of graphics	876
13.3.1	Graphic features	876
13.3.2	Parameters for changing features	876
13.3.3	Commands for global display features	881
13.4	Defining geometric objects without drawing them: <code>nodisp</code> . .	883
13.5	Geometric demonstrations: <code>assume</code>	885
13.6	Points in the plane	886
13.6.1	Points and complex numbers	886
13.6.2	The point in the plane: <code>point</code>	886
13.6.3	The difference and sum of two points in the plane: <code>+</code> <code>-</code> .	888
13.6.4	Defining random points in the plane: <code>point2d</code>	889
13.6.5	Points in polar coordinates: <code>polar_point</code> <code>point_polar</code>	889
13.6.6	Finding a point of intersection of two objects in the plane: <code>single_inter</code> <code>line_inter</code>	891
13.6.7	Finding the points of intersection of two geometric ob- jects in the plane: <code>inter</code>	892
13.6.8	Finding the orthocenter of a triangle in the plane: <code>orthocenter</code>	893
13.6.9	Finding the midpoint of a segment in the plane: <code>midpoint</code>	894
13.6.10	The barycenter in the plane: <code>barycenter</code>	894
13.6.11	The isobarycenter of n points in the plane: <code>isobarycenter</code>	895
13.6.12	The center of a circle in the plane: <code>center</code>	896
13.6.13	The vertices of a polygon in the plane: <code>vertices</code> <code>vertices_abc</code>	896
13.6.14	The vertices of a polygon in the plane, closed: <code>vertices_abca</code>	897
13.6.15	A point on a geometric object in the plane: <code>element</code> .	897
13.7	Lines in plane geometry	898
13.7.1	Lines and directed lines in the plane: <code>line</code>	898
13.7.2	Half-lines in the plane: <code>half_line</code>	900
13.7.3	Line segments in the plane: <code>segment</code> <code>Line</code>	900
13.7.4	Vectors in the plane: <code>segment</code> <code>vector</code>	901
13.7.5	Parallel lines in the plane: <code>parallel</code>	903
13.7.6	Perpendicular lines in the plane: <code>perpendicular</code> . . .	904
13.7.7	Tangents to curves in the plane: <code>tangent</code>	904
13.7.8	The median of a triangle in the plane: <code>median_line</code> .	905
13.7.9	The altitude of a triangle: <code>altitude</code>	906
13.7.10	The perpendicular bisector of a segment in the plane: <code>perpen_bisector</code>	906

13.7.11 The angle bisector: bisector	907
13.7.12 The exterior angle bisector: exbisector	908
13.8 Triangles in the plane	908
13.8.1 Arbitrary triangles in the plane: triangle	908
13.8.2 Isosceles triangles in the plane: isosceles_triangle	909
13.8.3 Right triangles in the plane: right_triangle	910
13.8.4 Equilateral triangles in the plane: equilateral_triangle	911
13.9 Quadrilaterals in the plane	912
13.9.1 Squares in the plane: square	913
13.9.2 Rhombuses in the plane: rhombus	914
13.9.3 Rectangles in the plane: rectangle	915
13.9.4 Parallelograms in the plane: parallelogram	916
13.9.5 Arbitrary quadrilaterals in the plane: quadrilateral	918
13.10 Other polygons in the plane	918
13.10.1 Regular hexagons in the plane: hexagon	918
13.10.2 Regular polygons in the plane: isopolygon	919
13.10.3 General polygons in the plane: polygon	920
13.10.4 Polygonal lines in the plane: open_polygon	921
13.10.5 Convex hulls: convexhull	922
13.11 Circles	923
13.11.1 Circles and arcs in the plane: circle	923
13.11.2 Circular arcs: arc	925
13.11.3 Circles (TI compatibility): Circle	926
13.11.4 Inscribed circles: incircle	926
13.11.5 Circumscribed circles: circumcircle	927
13.11.6 Excircles: excircle	927
13.11.7 The power of a point relative to a circle: powerpc	928
13.11.8 The radical axis of two circles: radical_axis	928
13.12 Other conic sections	929
13.12.1 The ellipse in the plane: ellipse	929
13.12.2 The hyperbola in the plane: hyperbola	930
13.12.3 The parabola in the plane: parabola	932
13.13 Coordinates in the plane	933
13.13.1 The affix of a point or vector: affix	933
13.13.2 The abscissa of a point or vector in the plane: abscissa	934
13.13.3 The ordinate of a point or vector in the plane: ordinate	935
13.13.4 The coordinates of a point, vector or line in the plane: coordinates	935
13.13.5 The rectangular coordinates of a point: rectangular_coordinates	938
13.13.6 The polar coordinates of a point: polar_coordinates	938
13.13.7 The Cartesian equation of a geometric object in the plane: equation	939
13.13.8 The parametric equation of a geometric object in the plane: parameq	939

13.14	Measurements	940
13.14.1	Measurement and display: <code>distanceat distanceatraw angleat angleatraw areaat areaatraw perimeterat perimeteratraw slopeat slopeatraw extract_measure</code>	940
13.14.2	The distance between objects in the plane: <code>distance</code>	942
13.14.3	The length squared of a segment in the plane: <code>distance2</code>	943
13.14.4	The measure of an angle in the plane: <code>angle</code>	943
13.14.5	The graphical representation of the area of a polygon: <code>plotarea areaplot</code>	945
13.14.6	The area of a polygon: <code>area</code>	946
13.14.7	The perimeter of a polygon: <code>perimeter</code>	947
13.14.8	The slope of a line: <code>slope</code>	948
13.14.9	The radius of a circle: <code>radius</code>	949
13.14.10	The length of a vector: <code>abs</code>	949
13.14.11	The angle of a vector: <code>arg</code>	950
13.14.12	Normalize a complex number: <code>normalize</code>	950
13.15	Transformations	950
13.15.1	General remarks	950
13.15.2	Translations in the plane: <code>translation</code>	951
13.15.3	Reflections in the plane: <code>reflection</code>	952
13.15.4	Rotation in the plane: <code>rotation</code>	953
13.15.5	Homothety in the plane: <code>homothety</code>	954
13.15.6	Similarity in the plane: <code>similarity</code>	955
13.15.7	Inversion in the plane: <code>inversion</code>	957
13.15.8	Orthogonal projection in the plane: <code>projection</code>	959
13.16	Properties	960
13.16.1	Checking if a point is on an object in the plane: <code>is_element</code>	960
13.16.2	Checking if three points are collinear in the plane: <code>is_collinear</code>	961
13.16.3	Checking if four points are concyclic in the plane: <code>is_concyclic</code>	962
13.16.4	Checking if a point is in a polygon or circle: <code>is_inside</code>	962
13.16.5	Checking if an object is an equilateral triangle in the plane: <code>is_equilateral</code>	963
13.16.6	Checking if an object in the plane is an isosceles tri- angle: <code>is_isosceles</code>	964
13.16.7	Checking if an object in the plane is a right triangle or a rectangle: <code>is_rectangle</code>	965
13.16.8	Checking if an object in the plane is a square: <code>is_square</code>	966
13.16.9	Checking if an object in the plane is a rhombus: <code>is_rhombus</code>	967
13.16.10	Checking if an object in the plane is a parallelogram: <code>is_parallelogram</code>	968

13.16.1	Checking if two lines in the plane are parallel: <code>is_parallel</code>	969
13.16.1	Checking if two lines in the plane are perpendicular: <code>is_perpendicular</code>	970
13.16.1	Checking if two circles in the plane are orthogonal: <code>is_orthogonal</code>	970
13.16.1	Checking if elements are conjugates: <code>is_conjugate</code>	971
13.16.1	Checking if four points form a harmonic division: <code>is_harmonic</code>	973
13.16.1	Checking if lines are in a bundle: <code>is_harmonic_line_bundle</code>	973
13.16.1	Checking if circles are in a bundle: <code>is_harmonic_circle_bundle</code>	974
13.17	Harmonic division	974
13.17.1	Finding a point dividing a segment in the harmonic ratio k : <code>division_point</code>	974
13.17.2	The cross ratio of four collinear points: <code>cross_ratio</code>	975
13.17.3	Harmonic division: <code>harmonic_division</code>	976
13.17.4	The harmonic conjugate: <code>harmonic_conjugate</code>	977
13.17.5	Pole and polar: <code>pole polar</code>	978
13.17.6	The polar reciprocal: <code>reciprocation</code>	979
13.18	Loci and envelopes	980
13.18.1	Loci: <code>locus</code>	980
13.18.2	Envelopes: <code>envelope</code>	983
13.18.3	The trace of a geometric object: <code>trace</code>	984
14	Three-dimensional Graphics	985
14.1	Introduction	985
14.2	Changing the view	986
14.3	The axes	986
14.3.1	Drawing unit vectors: <code>Ox_3d_unit_vector Oy_3d_unit_vector</code> <code>Oz_3d_unit_vector frame_3d</code>	986
14.4	Points in space	987
14.4.1	Defining a point in three-dimensions: <code>point</code>	987
14.4.2	Defining a random point in three-dimensions: <code>point3d</code>	988
14.4.3	Finding an intersection point of two objects in space: <code>single_inter line_inter</code>	989
14.4.4	Finding the intersection points of two objects in space: <code>inter</code>	990
14.4.5	Finding the midpoint of a segment in space: <code>midpoint</code>	992
14.4.6	Finding the barycenter of a set of points in space: <code>barycenter</code>	992
14.4.7	Finding the isobarycenter of a set of points in space: <code>isobarycenter</code>	993
14.5	Lines in space	993
14.5.1	Lines and directed lines in space: <code>line</code>	993
14.5.2	Half lines in space: <code>half_line</code>	995
14.5.3	Segments in space: <code>segment</code>	996

14.5.4	Vectors in space: vector	996
14.5.5	Parallel lines and planes in space: parallel	998
14.5.6	Perpendicular lines and planes in space: perpendicular	1000
14.5.7	Planes orthogonal to lines and lines orthogonal to planes in space: orthogonal	1002
14.5.8	Common perpendiculars to lines in space: common_perpendicular	1003
14.6	Planes in space	1004
14.6.1	Planes in space: plane	1004
14.6.2	The bisector plane in space: perpen_bisector	1005
14.6.3	Tangent planes in space: tangent	1006
14.7	Triangles in space	1007
14.7.1	Drawing triangles in space: triangle	1007
14.7.2	Isosceles triangles in space: isosceles_triangle	1008
14.7.3	Right triangles in space: right_triangle	1010
14.7.4	Equilateral triangles in space: equilateral_triangle	1012
14.8	Quadrilaterals in space	1013
14.8.1	Squares in space: square	1013
14.8.2	Rhombuses in space: rhombus	1014
14.8.3	Rectangles in space: rectangle	1016
14.8.4	Parallelograms in space: parallelogram	1018
14.8.5	Arbitrary quadrilaterals in space: quadrilateral	1019
14.9	Polygons in space	1019
14.9.1	Hexagons in space: hexagon	1020
14.9.2	Regular polygons in space: isopolygon	1020
14.9.3	General polygons in space: polygon	1022
14.9.4	Polygonal lines in space: open_polygon	1023
14.10	Circles in space: circle	1023
14.11	Conics in space	1025
14.11.1	Ellipses in space: ellipse	1025
14.11.2	Hyperbolas in space: hyperbola	1025
14.11.3	Parabolas in space: parabola	1026
14.12	Three-dimensional coordinates	1027
14.12.1	The abscissa of a three-dimensional point: abscissa	1027
14.12.2	The ordinate of a three-dimensional point: ordinate	1027
14.12.3	The cote of a three-dimensional point: cote	1027
14.12.4	The coordinates of a point, vector or line in space: coordinates	1028
14.12.5	The Cartesian equation of an object in space: equation	1029
14.12.6	The parametric equation of an object in space: parameq	1030
14.12.7	The length of a segment in space: distance	1030
14.12.8	The length squared of a segment in space: distance2	1031
14.12.9	The measure of an angle in space: angle	1031
14.13	Properties	1032

14.13.1	Checking if an object in space is on another object:	
	is_element	1032
14.13.2	Checking if points and/or lines in space are coplanar:	
	is_coplanar	1033
14.13.3	Checking if lines and/or planes in space are parallel:	
	is_parallel	1034
14.13.4	Checking if lines and/or planes in space are perpendicular:	
	is_perpendicular	1035
14.13.5	Checking if two lines or two spheres in space are orthogonal:	
	is_orthogonal	1035
14.13.6	Checking if points in space are collinear:	
	is_collinear	1036
14.13.7	Checking if points in space are concyclic:	
	is_concyclic	1037
14.13.8	Checking if points in space are cospherical:	
	is_cospherical	1038
14.13.9	Checking if an object in space is an equilateral triangle:	
	is_equilateral	1038
14.13.10	Checking if an object in space is an isosceles triangle:	
	is_isosceles	1039
14.13.11	Checking if an object in space is a right triangle or a rectangle:	
	is_rectangle	1040
14.13.12	Checking if an object in space is a square:	
	is_square	1040
14.13.13	Checking if an object in space is a rhombus:	
	is_rhombus	1041
14.13.14	Checking if an object in space is a parallelogram:	
	is_parallelogram	1042
14.14	Transformations in space	1043
14.14.1	General remarks	1043
14.14.2	Translation in space: translation	1043
14.14.3	Reflection in space with respect to a plane, line or point:	
	reflection symmetry	1045
14.14.4	Rotation in space: rotation	1046
14.14.5	Homothety in space: homothety	1047
14.14.6	Similarity in space: similarity	1048
14.14.7	Inversion in space: inversion	1049
14.14.8	Orthogonal projection in space: projection	1051
14.15	Surfaces	1051
14.15.1	Cones: cone	1051
14.15.2	Half-cones: half_cone	1052
14.15.3	Cylinders: cylinder	1053
14.15.4	Spheres: sphere	1053
14.15.5	The graph of a function of two variables: funcplot	1054
14.15.6	The graph of parametric equations in space: paramplot	1055
14.16	Solids	1055
14.16.1	Cubes: cube	1055
14.16.2	Tetrahedrons: tetrahedron pyramid	1058

14.16.3	Parallelepipeds: <code>parallelepiped</code>	1059
14.16.4	Prisms: <code>prism</code>	1060
14.16.5	Polyhedra: <code>polyhedron</code>	1060
14.16.6	Vertices: <code>vertices</code>	1061
14.16.7	Faces: <code>faces</code>	1061
14.16.8	Edges: <code>line_segments</code>	1062
14.17	Platonic solids	1062
14.17.1	Centered tetrahedra: <code>centered_tetrahedron</code>	1063
14.17.2	Centered cubes: <code>centered_cube</code>	1063
14.17.3	Octahedra: <code>octahedron</code>	1064
14.17.4	Dodecahedra: <code>dodecahedron</code>	1065
14.17.5	Icosahedra: <code>icosahedron</code>	1066
15	Multimedia	1069
15.1	Audio Tools	1069
15.1.1	Creating audio clips: <code>createwav</code>	1070
15.1.2	Reading <code>wav</code> files from disk: <code>readwav</code>	1071
15.1.3	Writing <code>wav</code> files to disk: <code>writewav</code>	1071
15.1.4	Audio playback: <code>playsnd</code>	1072
15.1.5	Averaging channel data: <code>stereo2mono</code>	1072
15.1.6	Audio clip properties: <code>channels</code> <code>bit_depth</code> <code>samplerate</code> <code>duration</code>	1073
15.1.7	Extracting samples from audio clips: <code>channel_data</code>	1073
15.1.8	Changing the sampling rate: <code>resample</code>	1074
15.1.9	Visualizing waveforms: <code>plotwav</code>	1075
15.1.10	Visualizing power spectra: <code>plotspectrum</code>	1076
15.1.11	Reading a <code>wav</code> file: <code>readwav</code>	1077
15.1.12	Writing a <code>wav</code> file: <code>writewav</code>	1078
15.1.13	Listening to a digital sound: <code>playsnd</code>	1078
15.1.14	Preparing digital sound data: <code>soundsec</code>	1078
15.2	Signal Processing	1079
15.2.1	Boxcar function: <code>boxcar</code>	1079
15.2.2	Rectangle function: <code>rect</code>	1080
15.2.3	Triangle function: <code>tri</code>	1080
15.2.4	Cardinal sine function: <code>sinc</code>	1081
15.2.5	Root mean square: <code>rms</code>	1081
15.2.6	Cross-correlation of two signals: <code>cross_correlation</code>	1082
15.2.7	Auto-correlation of a signal: <code>auto_correlation</code>	1083
15.2.8	Convolution of two signals or functions: <code>convolution</code>	1083
15.2.9	Low-pass filtering: <code>lowpass</code>	1086
15.2.10	High-pass filtering: <code>highpass</code>	1087
15.2.11	Apply a moving average filter to a signal: <code>moving_average</code>	1087
15.2.12	Performing thresholding operations on an array: <code>threshold</code>	1088
15.2.13	Bartlett-Hann window function: <code>bartlett_hann_window</code>	1091
15.2.14	Blackman-Harris window function: <code>blackman_harris_window</code>	1092

15.2.15 Blackman window function: <code>blackman_window</code>	1093
15.2.16 Bohman window function: <code>bohman_window</code>	1094
15.2.17 Cosine window function: <code>cosine_window</code>	1095
15.2.18 Gaussian window function: <code>gaussian_window</code>	1096
15.2.19 Hamming window function: <code>hamming_window</code>	1097
15.2.20 Hann-Poisson window function: <code>hann_poisson_window</code>	1098
15.2.21 Hann window function: <code>hann_window</code>	1099
15.2.22 Parzen window function: <code>parzen_window</code>	1100
15.2.23 Poisson window function: <code>poisson_window</code>	1101
15.2.24 Riemann window function: <code>riemann_window</code>	1102
15.2.25 Triangular window function: <code>triangle_window</code>	1103
15.2.26 Tukey window function: <code>tukey_window</code>	1104
15.2.27 Welch window function: <code>welch_window</code>	1105
15.2.28 An example: static noise removal by spectral subtraction	1106
15.3 Images	1108
15.3.1 Image structure in <code>Xcas</code>	1108
15.3.2 Reading images: <code>readrgb</code>	1109
15.3.3 Viewing images	1109
15.3.4 Creating or recreating images: <code>writergb</code>	1110
16 Using <code>giac</code> inside a program	1115
16.1 Using <code>giac</code> inside a C++ program	1115
16.2 Defining new <code>giac</code> functions	1116

Chapter 1

Index

Index

$\|$, 116
 $'$, 205
 $'*$, 241
 $'+$, 239
 $'-$, 241
 $'/'$, 241
 $()$, 97, 218, 453
 $*$, 173, 197, 241, 421, 448, 492, 496, 527
 $*=$, 102
 $+$, 128, 129, 173, 197, 239, 420, 448, 462, 494, 526, 888
 $+=$, 102
 $+\text{infinity}$, 96
 , , 461
 $-$, 173, 197, 241, 420, 448, 495, 526, 888
 $-=$, 102
 $->$, 109, 234
 $-\text{inf}$, 96
 $-\text{infinity}$, 96
 $.*$, 495, 529
 $.+$, 494, 526
 $.-$, 495, 526
 $...$, 445
 $./$, 495, 530
 $.^$, 530
 $.\text{xcasrc}$, 78
 $/$, 173, 197, 241, 423, 425, 448, 542
 $//$, 88, 838
 $/=$, 102
 $::=$, 517
 $::$, 87
 $:=$, 100, 109, 234, 516, 845
 $<$, 113
 $=$, 100
 $=<$, 101, 113, 517, 845
 $==$, 113
 $=>$, 100, 109, 323, 830, 845
 $>$, 113
 $>=$, 113
 $?.$, 114
 $[]$, 489
 $[[[]]$, 459
 $[]$, 459, 462, 463
 $\$$, 229, 454
 $\%$, 141, 229, 419, 433
 $\% \% \% \{ \% \% \}$, 351
 $\% \{ \% \}$, 98, 489
 $\%e$, 96
 $\%i$, 96
 $\%pi$, 96
 $\&*$, 527
 $\&\&$, 116
 $\&^$, 529
 \wedge , 173, 197, 424, 529
 $_$, 827, 833
 $!$, 158
 $!=$, 116
 $"$, 121
 m , 229, 260
 $\backslash n$, 122
 $\{ \}$, 835
 $a2q$, 609
 $abcuv$, 386
 about , 104, 335
 abs , 199, 246, 949
 abscissa , 934, 1027
 $\text{accumulate_head_tail}$, 720
 acos , 254
 acos2asin , 314
 acos2atan , 314
 acosh , 257
 acot , 254
 acsc , 254
 add , 479
 additionally , 104
 additionally , 105
 addtable , 336, 341

- adjoint_matrix, 557
- affix, 933
- Airy_Ai, 187
- Airy_Bi, 187
- algsbbs, 220
- algvar, 652
- altitude, 906
- and, 104, 116
- angle, 943, 1031
- angle_radian, 71
- angleat, 940
- angleatraw, 940
- animate, 702
- animate3d, 703
- animation, 704
- ans, 92
- append, 471
- apply, 484
- approx, 808
- approx_mode, 72
- arc, 925
- arccos, 254
- arccosh, 257
- archive, 103
- arcLen, 268
- arcsin, 254
- arcsinh, 257
- arctan, 254
- arctanh, 257
- area, 683, 946
- areaat, 940
- areaatraw, 940
- areaplot, 685, 945
- arg, 199, 950
- args, 855
- array, 323
- as_function_of, 261
- asc, 126
- asec, 254
- asin, 254
- asin2acos, 314
- asin2atan, 315
- asinh, 257
- assert, 840
- assign, 100
- assume, 104, 335, 836, 885
- at, 459, 460, 511
- atan, 254
- atan2acos, 316
- atan2asin, 315
- atanh, 257
- atrig2ln, 321
- augment, 472, 508
- auto_correlation, 1083
- autosimplify, 213
- axes, 665
- bar_plot, 723
- bareiss, 542
- bartlett_hann_window, 1091
- barycenter, 201, 894, 992
- base, 132
- basis, 545
- batons, 727
- begin, 838
- bernoulli, 170
- Beta, 185
- betad, 779
- betad_cdf, 779
- betad_icdf, 780
- bezier, 695
- Binary, 232
- binary, 130
- binomial, 159, 761
- binomial_cdf, 762
- binomial_icdf, 762
- bisection_solver, 819
- bisector, 907
- bit_depth, 1073
- bitand, 119
- bitor, 119
- bitxor, 119
- black, 663
- blackman_harris_window, 1092
- blackman_window, 1093
- BlockDiagonal, 503
- blockmatrix, 505
- blue, 663
- bohman_window, 1094
- boolean, 113
- border, 509
- box_constraints, 452
- boxcar, 1079
- boxwhisker, 497, 535, 717
- break, 854, 867
- breakpoint, 867

- brent_solver *brent_solver*, 819
- bvpsolve, 815
- cloc2, 194
- clop2, 194
- camembert, 724
- canonical_form, 206
- cap_flat_line, 663
- cap_round_line, 663
- cap_square_line, 663
- cas_setup, 78
- case, 850
- cat, 127, 129
- catch, 865
- cauchy, 782
- cauchy_cdf, 783
- cauchy_icdf, 783
- cauchyd, 782
- cauchyd_cdf, 783
- cauchyd_icdf, 783
- cd, 110
- cdf, 791
- ceil, 248
- Celsius2Fahrenheit, 831
- center, 896
- center2interval, 446
- centered_cube, 1063
- centered_tetrahedron, 1063
- cfactor, 72, 209
- cfsolve, 818, 821
- changebase, 544
- channel_data, 1073
- channels, 1073
- char, 126
- charpoly, 555
- chinrem, 387
- chisquare, 773
- chisquare_cdf, 773
- chisquare_icdf, 774
- chisquaret, 801
- cholesky, 564
- chrem, 149
- Ci, 178
- Circle, 926
- circle, 923, 1023
- circumcircle, 927
- classes, 718
- ClrDraw, 863
- ClrGraph, 863
- ClrIO, 843
- coeff, 355
- coeffs, 355
- col, 514
- colDim, 539
- coldim, 539
- collect, 361
- colNorm, 575
- colnorm, 575
- color, 881
- color*, 663
- , 819
- colspace, 547
- colSwap, 533
- colswap, 533
- comb, 159, 159
- combine, 331
- comDenom, 416
- comment, 88
- comments, 88, 838
- common_perpendicular, 1003
- companion, 559
- compare, 841
- complex, 841
- complex_mode, 72
- complex_variables, 73
- complexroot, 398
- concat, 472, 508
- cone, 1051
- confrac*, 166
- conic, 612
- conj, 200
- conjugate_equation, 305
- conjugate_gradient, 610
- cont, 867
- contains, 451, 490
- content, 360
- contourplot, 686
- convert, 132, 323, 354, 367, 452, 830
- convertir, 323, 367
- convex, 296
- convexhull, 922
- convolution, 1083
- coordinates, 935, 1028
- copy, 101, 846
- CopyVar, 103
- correlation, 725

- cos, **252**
- cos, **323, 331**
- cos2sintan, **318**
- cosh, **256**
- cosine_window, **1095**
- cot, **252**
- cote, **1027**
- count, **480, 533**
- count_eq, **478, 534**
- count_inf, **479, 535**
- count_sup, **479, 535**
- courbe_polaire, **696**
- covariance, **725**
- covariance_correlation, **725**
- cpartfrac, **416**
- crationalroot, **403**
- createwav, **1070**
- cross, **496**
- cross_correlation, **1082**
- cross_point, **663**
- cross_ratio, **975**
- crossP, **496**
- crossproduct, **496**
- csc, **253**
- cSolve, **619**
- CST, **108**
- cube, **1055**
- cumSum, **125, 481, 528**
- cumulated_frequencies, **721**
- curl, **294**
- curvature, **657**
- curve, **812**
- cyan, **663**
- cycle2perm, **192**
- cycleinv, **196**
- cycles2permu, **191**
- cyclotomic, **388**
- cylinder, **1053**
- dash_line, **663**
- dashdot_line, **663**
- dashdotdot_line, **663**
- dayofweek, **655**
- debug, **867**
- debugger, **838**
- default, **850**
- degree, **356**
- del, **107**
- delcols, **522**
- Delete, **232**
- DelFold, **112**
- delrows, **522**
- deltalist, **488**
- DelVar, **107**
- denom, **164, 414**
- densityplot, **688**
- derive, **271, 290**
- deriver, **271, 290**
- deSolve, **633**
- desolve, **633**
- Det, **436**
- det, **427, 542**
- det_minor, **543**
- dfc, **166**
- dfc2f, **169**
- diag, **503**
- diff, **271, 290**
- DIGITS, **70, 171, 808**
- Digits, **70, 171, 808**
- dim, **538**
- Dirac, **180**
- directories, **110**
- Disp, **843**
- DispG, **82, 863**
- DispHome, **863**
- display, **881**
- display, **663**
- distance, **942, 1030**
- distance2, **943, 1031**
- distanceat, **940**
- distanceatraw, **940**
- div, **140**
- divergence, **294**
- divide, **381**
- divis, **378**
- division_point, **974**
- divisors, **140**
- divpc, **439**
- dnewton_solverdnewton_solver, **823**
- do, **851**
- dodecahedron, **1065**
- DOM_COMPLEX, **841**
- DOM_FLOAT, **841**
- DOM_FUNC, **841**
- DOM_IDENT, **841**
- DOM_INT, **841**

- DOM_LIST, [841](#)
- DOM_RAT, [841](#)
- DOM_STRING, [841](#)
- DOM_SYMBOLIC, [841](#)
- domain, [262](#)
- dot, [496](#)
- dot_paper, [873](#)
- dotP, [496](#)
- dotprod, [496](#)
- double, [841](#)
- DrawFunc, [667](#)
- DrawParm, [692](#), [693](#)
- DrawPol, [696](#)
- DrawSlp, [681](#)
- droit, [618](#)
- DrwCtour, [686](#)
- dsolve, [633](#)
- duration, [1073](#)
- e, [96](#)
- e2r, [353](#)
- egcd, [385](#)
- egv, [551](#)
- egvl, [550](#)
- Ei, [175](#)
- eigenvals, [549](#)
- eigenvalues, [550](#)
- eigenvectors, [551](#)
- eigenvects, [551](#)
- eigVc, [551](#)
- eigVl, [550](#)
- element, [897](#)
- elif, [848](#)
- eliminate, [221](#)
- ellipse, [929](#), [1025](#)
- else, [848](#)
- end, [838](#), [848](#)
- end_for, [851](#)
- endfunction, [835](#)
- envelope, [983](#)
- epsilon, [650](#)
- epsilon2zero, [650](#)
- equal, [616](#)
- equal2diff, [617](#)
- equal2list, [617](#)
- equation, [939](#), [1029](#)
- equilateral_triangle, [911](#), [1012](#)
- erase, [872](#)
- erf, [181](#)
- erfc, [182](#)
- ERROR, [865](#)
- error, [865](#)
- euler, [155](#)
- euler_gamma, [96](#)
- euler_lagrange, [299](#)
- eval, [203](#)
- eval_level, [204](#)
- evala, [205](#)
- evalb, [119](#)
- evalc, [198](#)
- evalf, [106](#), [162](#), [171](#), [654](#), [808](#)
- evalm, [526](#)
- even, [143](#)
- evolute, [657](#)
- exact, [162](#), [654](#)
- exbisector, [908](#)
- excircle, [927](#)
- exp, [251](#)
- exp, [323](#), [331](#)
- exp2list, [117](#)
- exp2pow, [328](#)
- exp2trig, [316](#)
- expand, [206](#)
- expexpand, [325](#)
- expln, [323](#)
- exponential, [784](#)
- exponential_cdf, [784](#)
- exponential_icdf, [785](#)
- exponential_regression, [733](#)
- exponential_regression_plot, [733](#)
- exponentiald, [784](#)
- exponentiald_cdf, [784](#)
- exponentiald_icdf, [785](#)
- export_mathml, [84](#)
- EXPR, [841](#)
- expr, [129](#), [860](#)
- expression, [841](#)
- expression editor, [88](#)
- expression tree, [89](#)
- extract_measure, [940](#)
- extrema, [602](#)
- ezgcd, [383](#)
- f2nd, [165](#), [415](#)
- faces, [1061](#)
- Factor, [436](#)

- factor, **209**, **427**, **432**
- factor_xn, **359**
- factorial, **158**
- factoriser, **427**
- factors, **363**
- Fahrenheit2Celsius, **831**
- FALSE, **113**
- false, **113**
- falsepos_solver, **820**
- fclose, **858**
- fcoeff, **418**
- fdistrib, **206**
- feuille, **236**
- fieldplot, **698**
- filled, **663**
- findhelp, **62**
- fisher, **775**
- fisher_cdf, **775**
- fisher_icdf, **776**
- fisherd, **775**
- fitdistr, **795**
- flatten, **474**
- float2rational, **162**, **654**
- floor, **247**
- fMax, **269**
- fMin, **269**
- foldl, **487**
- foldr, **487**
- fopen, **858**
- for, **851**
- format, **862**
- fourier, **336**
- fourier_an, **332**
- fourier_bn, **333**
- fourier_cn, **334**
- fPart, **249**
- fprint, **858**
- frac, **249**
- fracmod, **425**
- frame_2d, **873**
- frame_3d, **986**
- frames, **877**
- frames, **702**
- frequencies, **720**
- frequencies_cumulees, **721**
- frequencies, **720**
- frobenius_norm, **573**
- from, **851**
- froot, **417**
- fsolve, **818**, **821**
- fullparfrac, **323**
- FUNC, **841**
- func, **841**
- funcplot, **667**, **1054**
- function, **835**
- function_diff, **267**
- fxnd, **165**, **415**
- Gamma, **183**
- gammad, **777**
- gammad_cdf, **777**
- gammad_icdf, **778**
- gauche, **618**
- gauss, **610**
- gauss_seidel_linsolve, **630**
- gaussian_window, **1096**
- gaussjord, **623**
- gaussquad, **811**
- gbasis, **408**
- Gcd, **133**, **381**, **382**, **435**
- gcd, **133**, **381**, **426**
- gcdex, **385**
- genpoly, **411**
- geometric, **780**
- geometric_cdf, **781**
- geometric_icdf, **782**
- getDenom, **164**, **414**
- GetFold, **112**
- getKey, **840**
- getNum, **164**, **413**
- getType, **841**
- GF, **429**
- giac, **53**, **55**
- gl_material, **663**
- gl_quaternion, **665**
- gl_rotation, **665**
- gl_shownames, **665**
- gl_texture, **665**, **877**
- gl_texture, **663**
- gl_x, **665**
- gl_x_axis_name, **665**
- gl_x_axis_unit, **665**
- gl_x_tick, **665**
- gl_y, **665**
- gl_y_axis_name, **665**
- gl_y_axis_unit, **665**

- gl_y_tick*, 665
- gl_z*, 665
- gl_z_axis_name*, 665
- gl_z_axis_unit*, 665
- gl_z_tick*, 665
- goto, 855
- grad, 290
- gramschmidt, 611
- Graph, 667
- graph2tex, 82
- graph3d2tex, 82
- graphe_suite, 697
- greduce, 409
- green, 663
- grid_paper, 875
- groupermu, 197
- hadamard, 529
- half_cone, 1052
- half_line, 900, 995
- halftan, 319
- halftan_hyp2exp, 320
- halt, 867
- hamdist, 121
- hamming_window, 1097
- hann_poisson_window, 1098
- hann_window, 1099
- harmonic_conjugate, 977
- harmonic_division, 976
- has, 653
- hasard, 739
- head, 124, 458
- Heaviside, 179
- hermite, 404
- hessenberg, 559
- hessian, 293
- heugcd, 383
- hexadecimal, 130
- hexagon, 918, 1020
- hidden_name, 663
- highpass, 1087
- hilbert, 504
- histogram, 719
- histogramme, 719
- hold, 205
- homothety, 954, 1047
- horner, 363
- hybrid_solverhybrid_solver, 823
- hybridsolverhybrids_solver, 823
- hybridsj_solverhybridsj_solver, 824
- hyp2exp, 325
- hyperbola, 930, 1025
- i, 96
- i[], 447
- iabcuv, 149
- ibasis, 545
- ibpdv, 286
- ibpu, 288
- icas, 53
- icdf, 792
- ichinrem, 149
- ichrem, 149
- icom, 157
- icosahedron, 1066
- id, 249
- identifer, 841
- identity, 502
- idivis, 140
- idn, 502
- iegcd, 148
- if, 848
- ifactor, 137
- ifactors, 138
- ifourier, 336
- IFTE, 115
- ifte, 114
- igamma, 184
- igcd, 133
- igcdex, 148
- ihermite, 561
- ilaplace, 641
- im, 197
- imag, 197
- image, 546
- implicitdiff, 274
- implicitplot, 689
- in, 867
- in_ideal, 410
- incircle, 926
- indets, 651
- inequationplot, 682
- inf, 96
- infinity, 96
- Input, 838

- input, 838
- InputStr, 838
- insert, 471
- inString, 127
- Int, 278
- int, 278
- intDiv, 140
- integer, 332, 841
- integer*, 106
- integrate, 278
- inter, 892, 990
- interactive_odeplot, 701
- interactive_plotode, 701
- internal directories, 112
- interp, 369
- intersect, 451, 491
- interval*, 323
- interval2center, 446
- inv, 425, 428, 542
- Inverse, 437
- inverse, 428
- inversion, 957, 1049
- invisible_point, 663
- invlaplace, 641
- invztrans, 649
- iPart, 247
- iquo, 140
- iquorem, 143
- iratecon, 425
- irem, 141
- is_collinear, 961, 1036
- is_concyclic, 962, 1037
- is_conjugate, 971
- is_coplanar, 1033
- is_cospherical, 1038
- is_cycle, 193
- is_element, 960, 1032
- is_equilateral, 963, 1038
- is_harmonic, 973
- is_harmonic_circle_bundle, 974
- is_harmonic_line_bundle, 973
- is_inside, 962
- is_isosceles, 964, 1039
- is_orthogonal, 970, 1035
- is_parallel, 969, 1034
- is_parallelogram, 968, 1042
- is_permu, 192
- is_perpendicular, 970, 1035
- is_prime, 145
- is_pseudoprime, 144
- is_rectangle, 965, 1040
- is_rhombus, 967, 1041
- is_square, 966, 1040
- ismith, 562
- isobarycenter, 895, 993
- isolve, 152
- isom, 578
- isopolygon, 919, 1020
- isosceles_triangle, 909, 1008
- isPrime, 145
- isprime, 145
- jacobi_equation, 304
- jacobi_linsolve, 629
- jacobi_symbol, 157
- jordan, 553
- JordanBlock, 504
- jusqua, 853
- kde, 792
- ker, 546
- kernel, 546
- kernel_density, 792
- kill, 867
- kolmogorovd, 788
- kolmogorovt, 803
- kovacicsols, 643
- l1norm, 493, 575
- l2norm, 493, 573, 575
- label, 855
- labels*, 665
- lagrange, 369
- lagrange*, 542
- laguerre, 405
- laplace, 341, 641
- laplacian, 291
- LaTeX, 82
- latex, 82
- lcm, 136, 384
- lcoeff, 357
- ldegree, 357
- left, 123, 236, 448, 467, 618
- legend, 881
- legend*, 665
- legendre, 403
- legendre_symbol, 156

- length, 123, 457
- lgcd, 136
- lhs, 618
- Li, 177
- ligne_polygonale, 728
- limit, 265
- lin, 326
- Line, 900
- line, 676, 898, 993
- line_inter, 891, 989
- line_paper, 874
- line_segments, 1062
- line_width_1, 663
- line_width_2, 663
- line_width_3, 663
- line_width_4, 663
- line_width_5, 663
- line_width_6, 663
- line_width_7, 663
- linear_interpolate, 728
- linear_regression, 731
- linear_regression_plot, 731
- LineHorz, 678
- LineTan, 679
- LineVert, 679
- linfnorm, 574
- linsolve, 626
- linsolve*, 542
- LIST, 841
- list*, 323
- list2exp, 118
- list2mat, 488
- listplot, 728
- lists, 463
- lll, 571
- ln, 252
- ln*, 323, 331
- lname, 651
- lncollect, 327
- lnexpand, 326
- load, 111
- local, 835
- locus, 980
- log, 252
- log*, 331
- log10, 252
- logarithmic_regression, 734
- logarithmic_regression_plot, 734
- logb, 252
- logistic_regression, 737
- logistic_regression_plot, 737
- loi_normal, 767
- lowpass, 1086
- lpsolve, 585
- LQ, 567
- LSQ, 631
- lsq, 631
- LU, 569
- lu, 567
- lvar, 652
- magenta, 663
- make_symbol, 859
- makelist, 464, 857
- makemat, 510
- makesuite, 462
- map, 484
- Maple, 86
- maple2xcas, 86
- maple_ifactors, 139
- markov, 797
- MAT, 841
- mat2list, 489
- MathML, 83
- mathml, 83
- matpow, 554
- matrix, 510
- matrix*, 323
- matrix_norm, 575
- max, 246
- maximize, 599
- maxnorm, 493, 574
- mean, 497, 535, 711
- median, 497, 535, 715
- median_line, 905
- member, 490
- mgf, 790
- mid, 124, 460
- midpoint, 450, 894, 992
- min, 246
- minimax, 608
- minimize, 599
- minor_det*, 542
- minus, 492
- mkisom, 579
- mksa, 830

- mod, [141](#), [433](#)
- modgcd, [383](#)
- mods, [141](#)
- moving_average, [1087](#)
- mRow, [531](#)
- mRowAdd, [532](#)
- mul, [482](#)
- mult_c_conjugate, [200](#)
- mult_conjugate, [207](#)
- multinomial, [765](#)
- mustache, [717](#)
- ncols, [539](#)
- nCr, [159](#)
- nDeriv, [810](#)
- negbinomial, [763](#)
- negbinomial_cdf, [764](#)
- negbinomial_icdf, [764](#)
- NewFold, [112](#)
- newList, [465](#)
- newMat, [503](#)
- newton, [809](#)
- newton_solvernewton_solver, [820](#)
- newtonj_solvernewtonj_solver, [823](#)
- nextperm, [189](#)
- nextprime, [147](#)
- nInt, [811](#)
- nlpsolve, [604](#)
- nodisp, [87](#), [883](#)
- noise removal, [1106](#)
- NONE, [841](#)
- nop, [462](#)
- nops, [457](#)
- norm, [493](#), [493](#), [573](#), [575](#)
- normal, [212](#), [419–421](#), [424](#)
- normal_cdf, [768](#)
- normal_icdf, [769](#)
- normald, [767](#)
- normald_cdf, [768](#)
- normald_icdf, [769](#)
- normalize, [200](#), [493](#), [950](#)
- normalt, [799](#)
- not, [116](#)
- nPr, [160](#)
- nprimes, [148](#)
- nrows, [538](#)
- nSolve, [818](#)
- nstep, [877](#)
- nstep, [667](#)
- nuage_points, [727](#)
- Nullspace, [547](#)
- nullspace, [546](#)
- NUM, [841](#)
- numdiff, [276](#)
- numer, [164](#), [413](#)
- octahedron, [1064](#)
- octal, [130](#)
- odd, [144](#)
- odeplot, [699](#)
- odesolve, [812](#), [814](#)
- op, [236](#), [462](#)
- open_polygon, [921](#), [1023](#)
- or, [104](#), [116](#)
- ord, [125](#)
- order_size, [439](#)
- ordinate, [935](#), [1027](#)
- orthocenter, [893](#)
- orthogonal, [1002](#)
- osculating_circle, [657](#)
- Output, [838](#)
- output, [838](#)
- Ox_2d_unit_vector, [873](#)
- Ox_3d_unit_vector, [986](#)
- Oy_2d_unit_vector, [873](#)
- Oy_3d_unit_vector, [986](#)
- Oz_3d_unit_vector, [986](#)
- p1oc2, [194](#)
- p1op2, [194](#)
- pa2b2, [152](#)
- pade, [443](#)
- parabola, [932](#), [1026](#)
- parallel, [903](#), [998](#)
- parallelepiped, [1059](#)
- parallelogram, [916](#), [1018](#)
- parameq, [939](#), [1030](#)
- paramplot, [692](#), [693](#), [1055](#)
- parfrac, [323](#)
- pari, [171](#)
- part, [222](#)
- partfrac, [416](#)
- partfrac, [323](#)
- parzen_window, [1100](#)
- Pause, [864](#)
- pcar, [555](#)
- pcar_hessenberg, [555](#)

- pcoef, 366
- pcoeff, 366
- perimeter, 947
- perimeterat, 940
- perimeteratraw, 940
- perm, 160
- perminv, 195
- permu2cycles, 190
- permu2mat, 192
- permuorder, 196
- perpen_bisector, 906, 1005
- perpendicular, 904, 1000
- peval, 359
- phi, 155
- pi, 96
- PIC, 841
- piecewise, 115
- piecewise defined functions, 114
- pivot, 625
- plane, 1004
- playsnd, 1072, 1078
- plot, 673
- plot3d, 674
- plotarea, 685, 945
- plotcontour, 686
- plotdensity, 688
- plotfield, 698
- plotfunc, 667
- plotimplicit, 689
- plotinequation, 682
- plotlist, 728
- plotode, 699
- plotparam, 692, 693
- plotpolar, 696
- plotseq, 697
- plotspectrum, 1076
- plotwav, 1075
- plus_point, 663
- pmin, 202, 556
- point, 886, 987
- point2d, 889
- point3d, 988
- point_milieu, 685
- point_point, 663
- point_polar, 889
- point_width_1, 663
- point_width_2, 663
- point_width_3, 663
- point_width_4, 663
- point_width_5, 663
- point_width_6, 663
- point_width_7, 663
- poisson, 766
- poisson_cdf, 766
- poisson_icdf, 767
- poisson_window, 1101
- polar, 978
- polar_coordinates, 938
- polar_point, 889
- polarplot, 696
- pole, 978
- poly1, 351
- poly2symb, 352
- polyEval, 359
- polygon, 920, 1022
- polygonplot, 728
- polyhedron, 1060
- polynom, 323, 367
- polynomial_regression, 736
- polynomial_regression_plot, 736
- poslbdLMQ, 401
- Postfix, 232
- posubLMQ, 400
- potential, 295
- pow2exp, 328
- power_regression, 735
- power_regression_plot, 735
- powermod, 424
- powerpc, 928
- powexpand, 327
- powmod, 424
- Prefix, 232
- prepend, 472
- preval, 222
- prevperm, 189
- prevprime, 147
- primpart, 360
- print, 843
- printpow, 844
- prism, 1060
- product, 482, 528, 529
- program, 837
- projection, 959, 1051
- proot, 824
- propFrac, 163
- propfrac, 163, 416

- Psi, 186
- psrgcd, 383
- ptayl, 364
- purge, 104, 107, 335, 837
- pwd, 110
- pyramid, 1058
- q2a, 609
- QR, 566
- qr, 565
- quadratic, 614
- quadrilateral, 918, 1019
- quadrant1, 663
- quadrant2, 663
- quadrant3, 663
- quadrant4, 663
- quantile, 497, 535, 716
- quartile1, 715
- quartile3, 715
- quartiles, 497, 535, 715
- quest, 92
- Quo, 378, 433
- quo, 378, 421
- quorem, 381, 422
- quote, 121, 205
- r2e, 352
- radical_axis, 928
- radius, 949
- rand, 739
- randbetad, 746
- randbinomial, 742
- randchisquare, 745
- randexp, 746
- randfisher, 745
- randgammad, 745
- randgeometric, 746
- randint, 739
- randmarkov, 798
- randMat, 757
- randmatrix, 757
- randmultinomial, 743
- randNorm, 744
- randnorm, 744
- random, 739
- random_variable, 747
- randperm, 189
- randpoisson, 743
- randPoly, 368
- randpoly, 368
- RandSeed, 742
- randseed, 742
- randstudent, 744
- randvar, 747
- randvector, 756
- range, 465
- rank, 544
- ranm, 369, 757
- rat_jordan, 551
- ratinterp, 377
- rational, 841
- rational_det, 542
- rationalroot, 402
- ratnormal, 215
- rdiv, 174
- re, 197
- read, 111
- readrgb, 1109
- readwav, 1071, 1077
- real, 197, 841
- realroot, 397
- reciprocation, 979
- rect, 1080
- rectangle, 915, 1016
- rectangle_droit, 685
- rectangle_gauche, 685
- rectangular_coordinates, 938
- red, 663
- REDIM, 523
- redim, 523
- reduced_conic, 612
- reduced_quadric, 614
- ref, 622
- reflection, 952, 1045
- regroup, 212
- Rem, 380, 434
- rem, 380, 422
- remain, 141
- remove, 470
- reorder, 369
- repeat, 853
- repeter, 853
- REPLACE, 525
- replace, 525
- resample, 1074
- residue, 443
- resoudre, 210, 290, 682

- restart, 107
- resultant, 393
- return, 835
- reverse_solve, 632
- revert, 442
- revlist, 474
- rhombus, 914, 1014
- rhombus_point, 663
- rhs, 618
- riemann_window, 1102
- right, 123, 236, 448, 467, 618
- right_triangle, 910, 1010
- risch, 281
- rm_a_z, 107
- rm_all_vars, 107
- rmbreakpoint, 867
- rmbrk, 867
- rms, 1081
- rmwatch, 869
- rmwtch, 867
- romberg, 811
- root, 175
- rootof, 365
- roots, 365
- rotate, 475
- rotation, 953, 1046
- round, 248
- row, 514
- rowAdd, 530
- rowDim, 538
- rowdim, 538
- rowNorm, 574
- rownorm, 574
- rowspace, 548
- rowSwap, 533
- rowswap, 533
- Rref, 438
- rref, 428, 623
- rsolve, 225
- sample, 739
- samplerate, 1073
- scalar_product, 496
- scalarProduct, 496
- SCALE, 531
- scale, 531
- SCALEADD, 532
- scaleadd, 532
- scatterplot, 727
- SCHUR, 559
- sec, 253
- secant_solversecant_solver, 820
- segment, 900, 901, 996
- select, 467
- semi_augment, 507
- seq, 454
- seq[], 97, 453
- seqplot, 697
- seqsolve, 224
- series, 439
- set[], 98, 489
- SetFold, 112
- shift, 475
- shift_phase, 311
- shuffle, 189
- Si, 178
- sign, 247
- signature, 195
- similarity, 955, 1048
- simp2, 165, 415
- simplex_reduce, 581
- simplify, 213, 313
- simult, 624
- sin, 252
- sin, 323, 331
- sin2costan, 317
- sinc, 1081
- sincos, 316
- sincos, 323
- single_inter, 891, 989
- sinh, 256
- size, 123, 457
- sizes, 463
- slope, 948
- slopeat, 940
- slopeatraw, 940
- smith, 563
- smod, 141
- snedecor, 775
- snedecor_cdf, 775
- snedecor_icdf, 776
- snedecord, 775
- solid_line, 663
- solve, 210, 290, 619, 682
- sommet, 236
- sort, 476

- SortA, 477
- SortD, 478
- soundsec, 1078
- specnorm, 575
- sphere, 1053
- spline, 372
- split, 208
- spreadsheet, 92
- sq, 250
- sqrfree, 362
- sqrt, 250
- square, 913, 1013
- square_point, 663
- srand, 742
- sst, 867
- sst_in, 867
- star_point, 663
- stdDev, 714
- stddev, 497, 535
- stddevp, 497, 714
- stdev, 713
- steffenson_solver, 821
- step, 851
- stereo2mono, 1072
- sto, 100
- Store, 100
- STR, 841
- string, 841, 862
- string, 323
- student, 770
- student_cdf, 771
- student_icdf, 772
- studentd, 770
- studentt, 800
- sturm, 389
- sturmab, 389
- sturmseq, 389
- subexpression, 90
- subexpressions, 89, 91
- subMat, 515
- subs, 218
- subsop, 468, 519
- subst, 216
- subtype, 841
- sum, 282, 479, 528
- sum_riemann, 284
- supposons, 104
- suppress, 469
- surd, 251
- svd, 571
- SVL, 570
- svl, 570
- swapcol, 533
- swaprow, 533
- switch, 850
- switch_axes, 872
- sylvester, 393
- symb2poly, 353
- symbol, 836
- symmetry, 1045
- syst2mat, 621
- table, 501
- tablefunc, 223
- tableseq, 227
- tabvar, 263
- tail, 124, 459
- tan, 252
- tan, 323
- tan2cossin2, 318
- tan2sincos, 317
- tan2sincos2, 318
- tangent, 680, 904, 1006
- tanh, 256
- taylor, 439
- tchebyshev1, 406
- tchebyshev2, 407
- tcoeff, 358
- tCollect, 312
- tcollect, 312
- tetrahedron, 1058
- TeXmacs, 54
- tExpand, 329
- texexpand, 329
- textinput, 838
- then, 848
- thickness, 877
- thiele, 375
- threshold, 1088
- throw, 865
- title, 665
- tlin, 310
- to, 851
- tpsolve, 597
- trace, 542, 984
- tran, 541

- translation, 951, 1043
- transpose, 541
- trapeze*, 685
- tri, 1080
- triangle, 908, 1007
- triangle_paper, 875
- triangle_point, 663
- triangle_window, 1103
- trig*, 331
- trig2exp, 320
- trigcos, 322
- trigexpand, 310
- triginterp, 371
- trigsimplify, 313
- trigsin, 321
- trigtan, 322
- trn, 544
- TRUE, 113
- true, 113
- trunc, 249
- truncate, 367
- try, 865
- tsimplify, 329
- tstep*, 877
- tukey_window, 1104
- tuple, 492
- type, 841
- ufactor, 832
- ugamma, 184
- unapply, 235
- unarchive, 103
- Unary*, 232
- unfactored, 689
- uniform, 759
- uniform_cdf, 760
- uniform_icdf, 760
- uniformd, 759
- uniformd_cdf, 760
- uniformd_icdf, 760
- union, 450, 491
- unit V, 200, 493
- unquote, 206
- Unquoted, 858
- until, 853
- user_operator, 232
- usimplify, 832
- ustep*, 877
- UTPC, 774
- UTPF, 776
- UTPN, 770
- UTPT, 772
- valuation, 357
- vandermonde, 505
- VAR, 841
- variable, 100
- variance, 497, 535, 712
- VARS, 107, 112
- VAS, 399
- VAS_positive, 400
- vector, 841, 901, 996
- vectors, 463
- version, 55
- vertices, 896, 1061
- vertices_abc, 896
- vertices_abca, 897
- vpotential, 295
- vstep*, 877
- WAIT, 864
- watch, 867
- weibull, 785
- weibull_cdf, 786
- weibull_icdf, 787
- weibulld, 785
- weibulld_cdf, 786
- weibulld_icdf, 787
- welch_window, 1105
- when, 114, 115
- while, 853
- white*, 663
- widget_size, 78
- wilcoxonp, 788
- wilcoxons, 789
- wilcoxont, 789
- write, 857
- writergb, 1110
- writewav, 1071, 1078
- wz_certificate, 160
- Xcas, 53, 57
- xcas.rc, 78
- xcas_mode, 71, 78
- xml_print, 84
- xor, 116
- xstep*, 877

xstep, 667
xyztrange, 78

yellow, 663
ystep, 877
ystep, 667

zeros, 210
Zeta, 187
zip, 486
zstep, 877
zstep, 667
ztrans, 648

Chapter 2

Introduction

2.1 Notations used in this manual

In this manual, the information that you enter is typeset in **typewriter font**. User input typically takes one of three forms:

- Commands that you enter on the command line.
For example, to compute the sin of $\pi/4$, you can type

`sin(pi/4)`

- Commands requiring a prefix key.
These are indicated by separating the prefix key and the standard key with a plus +. For example, to exit an **Xcas** session, you can type the control key along with the **q** key, which will be denoted

`Ctrl+Q`

- Menu commands.
When denoting menu items, submenus are connected using ►. For example, from within **Xcas** you can choose the **File** menu, then choose the **Open** submenu, and then choose the **File** item. This will be indicated by

`File ► Open ► File`

When describing entering a command, specific values that you enter for arguments are in typewriter font, while argument placeholders that should be replaced by actual values are in *italics*. Optional arguments will be enclosed by angle brackets. For example, you can find the derivative of an expression with the `diff` command (see Section 6.19.4 p.271), which takes the form `diff(expr⟨,x⟩/)` where *expr* is an expression and *x* is a variable or list of variables. If the optional variable is omitted, then it will default to **x**. A specific example is `diff(x*sin(x),x)`.

The index uses different typefaces for different parts of the language. The commands themselves are written with normal characters, command

options are written in *italics* and values of commands or options are written in **typewriter font**. For example (as you will see later), you can draw a blue parabola with the command

```
plotfunc(x^2,color = blue)
```

In the index, you will see

- `plotfunc`, the command, written in normal text.
- *color*, the command option, written in italics.
- **blue**, the value given to the option, written in typewriter font.

2.2 Interfaces for the `giac` library

The `giac` library is a C++ mathematics library. It comes with two interfaces you can use directly; a graphical interface and a command-line interface. All interfaces can do symbolic and numeric calculations, use `giac`'s programming language, and have a built in help function.

The graphical interface is called **Xcas**, and is the most full-featured interface. **Xcas** has additional help features to make it easy to use, plus it has a built-in spreadsheet, it can do dynamic geometry and it can do turtle graphics. The output given by this interface is typeset; for example:

Input:

```
sqrt(1/2)
```

Output:

$$\frac{\sqrt{2}}{2}$$

The command-line interface can be run inside a terminal, and in a graphical environment can also draw graphs. The output given by this interface is in text form; for example:

Input:

```
sqrt(1/2)
```

Output:

```
sqrt(2)/2
```

There is also a web version, which can be run through a javascript-enabled browser (it works best with Firefox), either over the internet or from local files. Other programs (for example, **TeXmacs**) have interfaces for the command-line version. Some of these interfaces, such as the two mentioned here, typeset their output.

2.2.1 The Xcas interface

How you start **Xcas** in a graphical environment depends on which operating system you are using.

- If you are using Unix, you can usually find an entry for the program in a menu provided your desktop environment. Otherwise, you can start it from a terminal by typing

```
xcas &
```

If for some reason **Xcas** becomes unresponsive, you can open a terminal and type

```
killall xcas
```

This will kill any running **Xcas** processes. **Xcas** keeps an automatic backup files, so when you restart **Xcas**, you will be asked if you want to resume where you left off.

- If you are running Windows, you can use the explorer to go to the directory where **Xcas** is installed. In that directory is a file called **xcas.bat**. You can click on that file to start **Xcas**.
- If you are running Mac OS, you can use the Finder to go to the **xcas_image.dmg** file and double-click it. Then double-click the **Xcas** disk icon. Finally, you can double-click the **Xcas** program to launch **Xcas**.

When you start **Xcas**, a window will open with menu entries across the top, below that will be a bar giving information about the current **Xcas** configuration, and below that will be an entry line you can use to enter commands. This interface will be described in more detail later, but the menu item

Help►Interface

will bring up an introduction.

2.2.2 The command-line interface: icas giac

In Unix and MacOS you can run **giac** from a terminal with the command **icas** (the command **giac** also works). There are two ways to use the command-line interface.

If you just want to evaluate one expression, you can give **icas** the expression (in quotes) as a command line argument. For example, to factor the polynomial $x^2 - 1$, you can type

```
icas 'factor(x^2-1)'
```

at a command prompt. The result will be

```
(x-1)*(x+1)
```

and you will be returned to the operating system command line.

If you want to evaluate several commands, you can enter an interactive **giac** session by entering the command **icas** (or **giac**) by itself at a command prompt. You will then be given a prompt specifically for **giac** commands, which will look like

0»

You can enter a **giac** command at this prompt and get the result.

```
0>> factor(x^2-1)
(x-1)*(x+1)
1>>
```

After the result, you will be given another prompt for **giac** commands. You can exit this interactive session by typing **Ctrl+D**.

You can also run **icas** in batch mode; that is, you can have **icas** run **giac** commands stored in a file. This can be done in Windows as well as Unix and Mac OS. To do this, simply enter

icas filename

at a command prompt, where *filename* is the name of the file containing the **giac** commands.

2.2.3 The Firefox interface

You can run **giac** without installing it by using a javascript-enabled web browser. Using Firefox for this is highly recommended; Firefox runs **giac** several times faster than Chrome, for example, and Firefox also supports MathML natively.

To run **giac** through Firefox, you can open the url <https://www-fourier.ujf-grenoble.fr/~parisse/giac/xcasen.html>. At the top of this page will be a button which will open a quick tutorial; the tutorial also tells you how to install the necessary files to run **giac** through Firefox without being connected to the internet.

2.2.4 The TeXmacs interface

TeXmacs (<http://www.texmacs.org>) is a sophisticated word processor with special mathematical features. As well as being designed to nicely typeset mathematics, it can be used as a frontend for various mathematics programs, including **giac**.

Once you've started TeXmacs, you can interactively run **giac** within TeXmacs with the menu command **Insert►Session►Giac**. Once started, you can enter **giac** commands as you would in the command-line interface. You can later re-enter a **giac** entry line by choosing it with your arrow keys or clicking on it with a mouse. The TeXmacs interface also has a menu containing **giac** commands.

Note that a **giac** session in TeXmacs may be started directly from a terminal using the script **xgiac** available in the **src** subdirectory of the **giac** source.

Within TeXmacs, you can combine `giac` commands and their output with ordinary text. To enter normal text within a `giac` session, use the menu item **Focus►Insert Text Field Above**.

You can also use mathematical input when entering commands, which allows for fast typing of complex formulas. For details, see the `giac` plugin documentation in TeXmacs. Also, any command output in a TeXmacs session (or a part of it) may be copied back to an input field, thanks to the conversion routines between `giac` syntax and TeXmacs Scheme.

Graphics created by `giac` are converted to PDF and embedded into TeXmacs sessions. Note that this requires the Ghostscript package to be installed on your system.

2.2.5 Checking the version of `giac` that you are using: `version giac`

The `version` (or `giac`) command returns the version of `giac` that is running. It doesn't have any arguments, but it does require parentheses.

Input:

```
version()
```

Output:

```
"giac 1.6.0, (c) B. Parisse and R. De Graeve, Institut Fourier, Universite de  
Grenoble I"
```


Chapter 3

The Xcas interface

3.1 The entry levels

The **Xcas** interface can run several independent calculation sessions, each session will be contained in a separate tab. Before you understand the **Xcas** interface, it would help to be familiar with the components of a session.

Each session can have any number of input levels. Each input level will have a number to the left of it; the number is used to identify the level. Each level can have one of the following:

- A command line.

This is the default; you can open a new command line with **Alt+N**.

You can enter a **giac** command (or a series of commands separated by semicolons) on a command line and send it to be evaluated by hitting enter. The result will then be displayed, and another command line will appear. You can also scroll through the command history with **Ctrl+Up** and **Ctrl+Down**.

If the output is a number or an expression, then it will appear in blue text in a small area below the input region; this area will be an expression editor (see Section 4.3 p.88). There will be a scrollbar and a small **M** to the right of this area; the **M** is a menu which gives you various options.

If the output is a graphic, then it will appear in a graphing area below the input region. To the right of the graphic will be a control panel which you can use to manipulate the graphic (see Section 8.2 p.662).

- An expression editor.

See Section 4.3 p.88. You can open an expression editor with **Alt+E**.

- A two-dimensional geometry screen.

See Section 8.2 p.662. You can open a two-dimensional geometry screen with **Alt+G**. This level will have a screen, as well as a control panel, menus and a command line to control the screen.

- A three-dimensional geometry screen.

See Section 8.2 p.662. You can open a three-dimensional geometry

screen with **Alt+H**. This level will have a screen, as well as a control panel, menus and a command line to control the screen.

- A turtle graphics screen.
You can open a turtle graphics screen with **Alt+D**. This level will have a screen, as well as a program editor and command line.
- A spreadsheet.
See Section 4.5 p.92. You can open a spreadsheet with **Alt+T**. A spreadsheet can open a graphic screen.
- A program editor.
See Section 12.1.1 p.835. You can open a program editor with **Alt+P**.
- A comment line.
See Section 4.2 p.88. You can open a comment line with **Alt+C**.

Levels can be moved up and down in a session, or even moved to a different session.

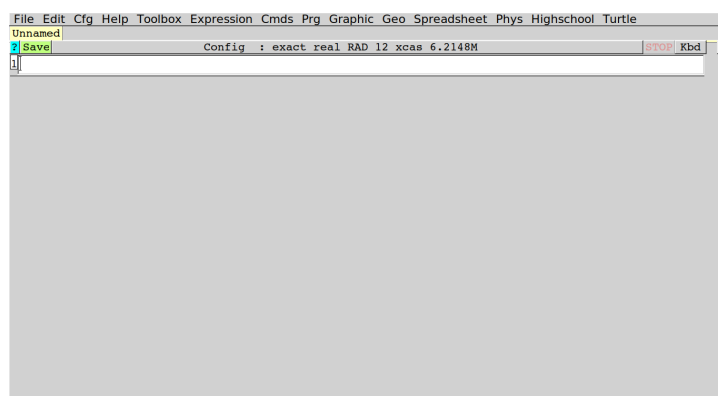
The level containing the cursor is the *current level*. The current level can be evaluated or re-evaluated by typing enter.

You can select a level (for later operations) by clicking on the number in the white box to the left of the level. Once selected, the box containing the number turns black. You can select a range of levels by clicking on the number for the beginning level, and then holding the shift key while you click on the number for the ending level.

You can copy the instructions in a range of levels by selecting the range, and then clicking the middle mouse button on the number of the target level.

3.2 The starting window

When you first start **Xcas**, you get a largely blank window.





The first row will consist of the main menus; you can save and load **Xcas** sessions, configure **Xcas** and its interface and run various commands with entries from these menus.

The second row will contains tabs; one tab for each session that you are running in **Xcas**. Each tab will have the name of its session, or **Unnamed** if

the session has no name. The first time you start **Xcas**, there will be only one session, which will be unnamed.

The third row will contain various buttons.

- The first button, , opens the help index (The same as the **Help►Index** menu entry; see Section 3.3 p.61). If there is a command on the command line, the help index will open at this command.
- The second button, , saves the session in a file. The first time you click on it you will be prompted for a file name ending in **.xws** in which to save the session. The button will be pink if the session is not saved or if it has changed since the last change, it will be green once the session is saved. The name in the title will be the name of the file used to save the session.
- The third button, which in the picture above is

Config: exact real RAD 12 xcas 6.2148M,

is a status line indicating the current **Xcas** configuration (see Section 3.5 p.70). If the session is unsaved, it will begin with **Config:**; if the session is saved in a file *filename.xws*, this button will begin with **Config filename.xws:**. Other information on this status line:

1. **exact** or **approx**

This tells you whether **Xcas** will give you exact values, such as $\sqrt{2}$, when possible or gives you decimal approximations, such as 1.4142135. (See Section 3.5.4 p.72.)

2. **real**, **cplx** or **CPLX**.

When this shows **real** (for example), then **Xcas** will by default only find real solutions of equations. When this shows **cplx**, then **Xcas** will find complex solutions of equations. When this shows **CPLX**, then **Xcas** will regard variables as complex; for example, it won't simplify **re(z)** (the real part of the variable *z*) to *z*. (See sections 3.5.5 and 3.5.6.)

3. **RAD** or **DEG**.

This tells you whether angles, as in trigonometric arguments, are measured in radians or degrees. (See Section 3.5.3 p.71.)

4. An integer.

This tells you how many significant digits will be used in floating point calculations. (See Section 3.5.1 p.70.)

5. **xcas**, **python ^==***, **python ^=xor**, **maple**, **mupad**, or **ti89**.

This tells you what syntax **Xcas** will use. **Xcas** can be set to emulate the languages of Python, Maple, MuPAD, or the TI89 series of calculators. (See Section 3.5.2 p.71.)

6. The last item tells you how much memory **Xcas** is using.

Clicking on this status line button opens a window where you can configure the settings shown on this line as well as some other settings; you can also open the window with the menu item **Cfg►CAS Configuration** (see Section 3.5.7 p.73).

- The fourth button, **STOP** (in red), is used to halt a computation which is running on too long.
- The fifth button, **Kbd**, toggles an on-screen scientific keyboard at the bottom of the window.

x	y	'	*	[]	{	}	:	oo		inv	+	7	8	9	esc	X
z	t	:	=	()	,)		i	sqrt	>	-	-	4	5	6	b7l
~	=>	factor		a	sin	a	cos	a	tan	^	*	1	2	3		paste	abc
simpli	prg	lim		ln	exp	log10	10^	%	/	0	.	E					

Along the right hand side of the keyboard are some keys that can be used to change the keyboard.

- The **X** key hides the keyboard, just like pressing the **Kbd** button again.
- The **cmds** key toggles a menu bar at the bottom of the screen which can be used as an alternate menu or persistent submenu. This bar will contain buttons **home**, **<<**, some menu titles, **>>**, **var**, **cust** and **X**.

The **<<** and **>>** buttons scroll through menu items. Clicking on one of the menu buttons will perform the appropriate action or replace the menu items by its submenu items. When submenu items appear, there will also be a **BACK** button to return to the previous menu. Clicking on the **home** button returns the menu buttons to the main menu.

After the menu buttons is a **var** button. This replaces the menu buttons by buttons representing the variables that you have defined. After that is a **cust** button, which displays commands that you store in a list variable **CST** (see section 5.4.10).

The last button, **X**, closes the menu bar.

- The **msg** key brings up a message window at the bottom of the window which will give you helpful messages; for example, if you save a graphic, it will tell you the name of the file it is saved in and how to include it in a **L^AT_EX** file.
- The **abc** key toggles the keyboard between the scientific keyboard and an alphabetic keyboard.
- The fifth button, **X**, closes the current session.

3.3 Getting help

Xcas is an extensive program, but using it is simplified with several different ways of getting help. The help menu (see section 3.4.4) has several submenus for various forms of help, some of which are mentioned below.

Tooltips

If you hover the mouse cursor over certain parts of the **Xcas** window, a temporary window will appear with information about the part. For example, if you move the mouse cursor over the status line, you will get a message saying **Current CAS status. Click to modify.**


If you type a function name in the **Xcas** command line, a similar temporary window will appear with information about the function.

HTML help

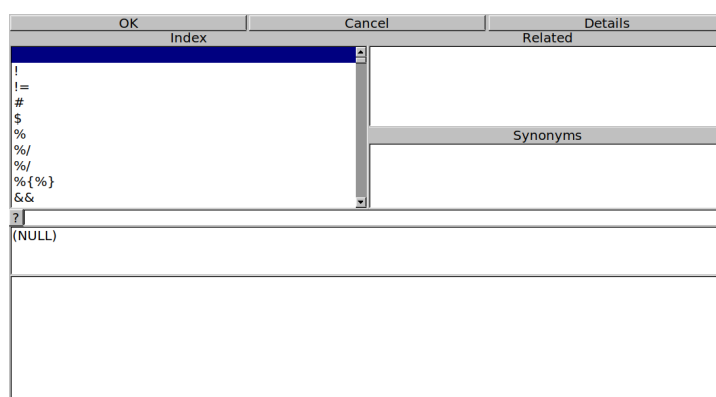
If you press the **F12** key, you will get a window which you can use to search the html version of the manual. You can also open this window with the menu entry **Help►Find word in HTML help**.

The HTML help window has a search area; if you type a string in that area you will be given a list of help topics that contain that string. If you choose a topic and click **View**, your web browser will show the appropriate page of the manual.

The help index

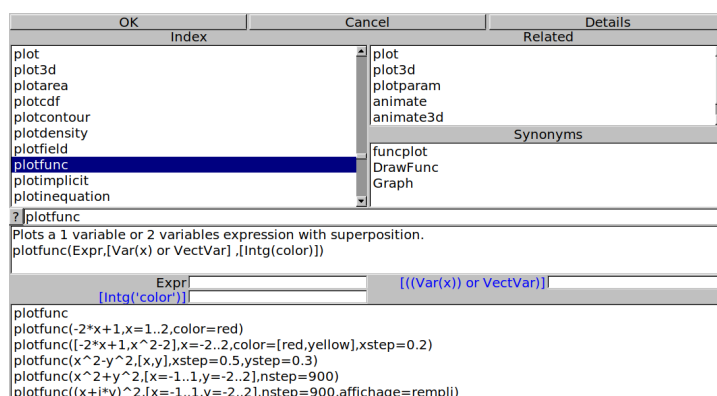
If you click on the  button on the status line you will get the help index. You can also get the help index with the menu item **Help►Index**.

The help index is a list of the **giac** function and variable names.



You can scroll through the help index items and click on the word that you want. There is also a line in the help index window that you can use to search the index; you can enter some text and be taken to the part of the index with the words that begin with that text. The **?** button next to this search line will open the HTML help window.

If you select a function or variable name, a list of related words (names of functions or variables) and a list of synonymous words will appear in regions to the right.



Below the search line, there will be an area which will have a brief description of the chosen term as well as how to call it. If the term is a command name, the calling sequence will be given as the command name with the arguments within parentheses separated by commas. Any optional arguments will be shown within brackets. In the above example, the first argument to `plotfunc` is an expression, representing the function to be graphed. There is an optional second argument, which is either a variable name (which defaults to `x`) or a vector of variable names for multivariable functions. Finally, there is an optional third argument which can be used to specify a color for the graph.

Below the brief description will be some entry fields that you can use to enter the arguments. If you fill them out and press the enter key, the command with the arguments filled out will be put on the command line.

Below the entry fields for the arguments will be a list of examples of the command being used. If you click on one of these examples, it will be put on the command line.

A more thorough description of the function and its arguments is available with the **Details** button at the top of the help index, which will open the relevant part of the manual in your browser. Alternatively, if you click on the `[?]` button next to the search line, you will be taken to the HTML help window.

You can also open the help index in the following ways:

- You can press the tab key while at the **Xcas** command line.
If you have entered part of a command name, you will be at the part of the index with words that begin with the text that you entered.
- You can select a command from one of the menus. If **Auto index help** is chosen (see Section 3.5.9 p.77), then the help index will open with the command chosen.

findhelp

You can get help from **Xcas** by using the `findhelp` function. If you enter `findhelp(function)` (or equivalently `?function`) at the command input, where *function* is the name of a **giac** function, then some notes on *function* will appear in the answer portion and the appropriate page of the manual will appear in your web browser.

3.4 The menus

The menus provide different ways to work with **Xcas** and its sessions, as well as ways of inserting functions and constants into the current session. Selecting a menu item corresponding to a function or constant brings up the help index (see section 3.3) with the chosen function or constant selected.

3.4.1 The File menu

The **File** menu contains commands that are used to save sessions, save parts of sessions, and load previously saved sessions. This menu contains the following entries:

- **New Session**
This creates and opens a new session.
The new session will be in a new tab, which will be labeled **Unnamed** until you save it (using the menu item **File►Save** or the keystroke **Alt+S**).
- **Open**
This allows you to open a previously saved session.
There will be a submenu with a list of saved session files in the primary directory (see Section 5.6.1 p.110) that you can open, as well as a **File** item which will open a directory browser you can use to find a session file. This directory browser can also be opened with **Alt-O**.
- **Import**
This allows you to open a session that was created with the Maple CAS, a TI89 calculator or a Voyage200 calculator. You can execute this session with the **Edit►Execute Session** menu entry, but it may be better to execute the commands one at a time to see if any modifications need to be done.
- **Clone**
This creates a copy of the current session in a Firefox interface; either using the server at <http://www-fourier.ujf-grenoble.fr/~parisse/xcasen.html> (Online) or a local copy (Offline).
- **Insert**
This allows you to insert a previously saved session, a link to a Firefox session, or a previously saved figure, spreadsheet or program.
- **Save (Alt+S)**
This saves the current session.
- **Save as**
This saves the current session under a name that you choose.
- **Save all**
This saves all of the sessions.

- **Export as**
This allows you to save the current session in different formats; either in **KhiCas** (which is **giac** ported to run on various calculators) format, standard **Xcas** format, **Xcas** with Python syntax format, Maple format, MuPAD format or TI89 format.
- **Kill**
This kills the current session.
- **Print**
This allows you to create an image of the session in various ways. The **Preview** menu item saves an image of the current session in a file that you name. The **To printer** item sends an image of the current session to the printer. The **Preview selected levels** item saves the images of the commands and outputs of the selected levels, each in a separate file.
- **LaTeX**
This has submenu items that render the session in **L^AT_EX** and give you the result in various ways. The **LaTeX preview** menu item displays a compiled **L^AT_EX** version of the current session. The **LaTeX print** item saves a copy of the session in **L^AT_EX** form, along with the compiled version in various formats. The **LaTeX print selection** does the same as **LaTeX print**, but only for the selected levels.
- **Screen capture**
This creates a screenshot that is saved in various formats.
- **Quit and update Xcas**
This quits **Xcas** after checking for a newer version.
- **Quit (Ctrl+Q)**
This quits **Xcas**.

3.4.2 The Edit menu

The **Edit** menu contains commands that are used to execute and undo parts of the current session. This menu contains the following entries:

- **Execute worksheet (Ctrl-F9)**
This recalculates each level in the session.
- **Execute worksheet with pauses**
This recalculates each level in the session, pausing between calculations.
- **Execute below**
This recalculates the current level and each level below it.
- **Remove answers below**
This removes the answers to the current level and the levels below it.

- **Undo (Ctrl+Z)**
This undoes the latest edit done to the levels, including a deletion of a level. It can be repeated to undo more than one edit.
- **Redo (Ctrl+Y)**
This redoes the undone editing.
- **Paste**
This pastes the contents of the system clipboard to the cursor position.
- **Del selected levels**
This deletes any entry levels that you have selected.
- **selection -> LaTeX (Ctrl+T)**
This puts a \LaTeX version of the selection (level, part of a level, or answer selected by clicking and dragging the mouse) on the system clipboard.
- **New entry (Alt+N)**
This inserts a new entry level above the current one.
- **New parameter (Ctrl+P)**
This brings up a window in which you can enter a name and conditions for a new parameter.
- **Insert newline**
This inserts a newline below the cursor. Note that simply typing return will evaluate the current entry rather than inserting a newline.
- **Merge selected levels**
This merges the selected levels into a single level.

3.4.3 The Cfg menu

The **Cfg** menu contains commands that are used to set the behaviour of **Xcas**. This menu contains the following entries:

- **Cas configuration**
This opens a window that allows you to configure how **Xcas** performs calculations (see Section 3.5.7 p.73). This is the same window you get when you click on the status line.
- **Graph configuration**
This opens a window that allows you to configure the default settings for a graph (see Section 3.5.8 p.76). This includes such things as the initial ranges of the variables. Each graph also has a **cfg** button to configure the settings on a per graph basis.
- **General configuration**
This opens a window that allows you to configure various non-computational aspects of **Xcas**, such as the fonts, the default paper size, and the like (see Section 3.5.9 p.77).

- **Mode (syntax)**
This changes the default syntax (see Section 3.5.2 p.71). By default, Xcas uses its own syntax, but you can change it to Python syntax, Maple syntax, MuPAD syntax or TI89 syntax.
- **Show**
This displays parts of Xcas.
 - **DispG**
This shows the graphics display screen; which has all graphical commands from the session together on one screen.
 - **keyboard**
This shows the on-screen keyboard; the same as clicking on the **Kbd** button on the status line (see Section 3.2 p.58, item 3.2).
 - **bandeau**
This shows the menu buttons at the bottom of the window; the same as clicking on **cmds** on the on-screen keyboard (see Section 3.2 p.58, item 3.2).
 - **msg**
This shows the messages window; the same as clicking on **msg** on the on-screen keyboard (see Section 3.2 p.58, item 3.2).
- **Hide**
This hides the same items that you can show with **Show**.
- **Index language**
This allows you to choose a language in which to display the help index.
- **Colors**
This allows you to choose colors for various parts of the display.
- **Session font**
This allows you to choose a font for the sessions.
- **All fonts**
This allows you to choose fonts for the session, the main menu and the keyboard.
- **browser**
This allows you to choose a browser that Xcas will use when needed. If this is blank, then Xcas will use its own internal browser.
- **Save configuration**
This saves the configurations that you chose with the **Cfg** menu or chose by clicking on the status line.

3.4.4 The Help menu

The **Help** menu contains commands that let you get information about Xcas from various sources. This menu contains the following entries:

- **Index**
This brings up the help index (see Section 3.3 p.61).
- **Find word in HTML help (F12)**
This brings up a page which helps you search for keywords in the html documentation that came with Xcas (see Section 3.3 p.61). The help will be displayed in your browser.
- **Interface**
This brings up a tutorial for the Xcas interface. The tutorial will be displayed in your browser.
- **Reference card, fiches**
This brings up a pdf reference card for Xcas. The card will be displayed in your browser.
- **Manuals**
This allows you to choose from a variety of manuals for XCAS, which will appear in your browser.
 - **CAS reference**
This brings up the manual for Xcas.
 - **Algorithmes (HTML)**
This brings up a manual for the algorithms used by Xcas.
 - **Algorithmes (PDF)**
This brings up a pdf version of the manual for the algorithms used by Xcas.
 - **Geometry**
This brings up a manual for two-dimensional geometry in Xcas.
 - **Programmation**
This brings up a manual for programming in Xcas.
 - **Simulation**
This brings up a manual for statistics and using the Xcas spreadsheet.
 - **Turtle**
This brings up a manual for using the Turtle drawing screen in Xcas.
 - **Exercices**
This brings up a page of exercises that you can do with Xcas.
 - **Amusement**
This brings up a page of mathematical amusements that you can work through with Xcas.
 - **PARI-GP**
This brings up documentation for the GP/PARI functions.
- **Internet**
The **Internet** menu contains menu items that take you to various web pages related to Xcas. Among them are the following entries:

- **Forum**
This takes you to the **Xcas** forum.
- **Update help**
This installs updated help files (retrieved from the **Xcas** website).

There are also several menu items that take you to **Xcas** related pages written in French; namely:

- **Aide-memoire lycee**
This takes you to a paper discussing **Xcas** and high school.
- **Documents pedagogiques lycee**
This takes you to a page on the **Xcas** website with a list of useful links.
- **Documents algorithmique**
This takes you to a page on the **Xcas** website with a list of links.
- **Site Lycee de G. Connan**
This takes you to a page about a free book written by Guillaume Connan teaching algorithms to high school students.
- **Site Lycee de L. Briel**
This takes you to a website about **Xcas** for high school students.
- **Calcul formel au lycee, par D. Chevallair**
This takes you to a pdf file discussing the use of **Xcas** in high school.
- **Site de F. Han**
This takes you to a website by Frederic Han about **Xcas** and a QT frontent for **giac**.
- **Ressources Capes**
This takes you to a website with various external sources.
- **Ressources Agregation externe**
This takes you to a collection of external resources.
- **Ressources Agregation interne**
This takes you to a page on the **Xcas** website.

- **Start with CAS**

This menu has the following entries:

- **Tutorial**
This opens up the tutorial.
- **Solutions**
This opens up the solutions to the exercises in the tutorial.

- **Tutoriel algo**

This opens up a tutorial on algorithms and programming with **Xcas**.

- **Rebuild help cache**

This rebuilds the help index.

- **About**
This displays a message window with information about **Xcas**.
- **Examples**
This allows you to choose from a variety of example worksheets, which will then be copied to your current directory and opened.

3.4.5 The Toolbox menu

The **Toolbox** menu contains commands that are used to insert operators into the session. This menu includes the following entries:

- **New entry (Alt+N)**
This inserts a new level.
- **New comment (Alt+C)**
This inserts a new comment level.

The other entries let you insert mathematical operations into the current level. If **Auto index help** is chosen (see Section 3.5.9 p.77), then the help index will open help index (see Section 3.3 p.61) with the chosen command selected.

3.4.6 The Expression menu

The **Expression** menu contains commands that are used to transform expressions. The first entry is **New expression** (which is equivalent to **Alt+E**), which inserts a new level and brings up the on-screen keyboard (see Section 3.2 p.58, item 3.2). The rest of the entries can be used to insert various transformations.

3.4.7 The Cmds menu

The **Cmds** menu contains various **giac** functions and constants separated into categories. If **Auto index help** is chosen (see Section 3.5.9 p.77), then when you select a function or constant, the help index (see Section 3.3 p.61) opens with the function or constant selected, which can be used to insert the entry on the command line. Otherwise, the constant or function will be inserted on the command line.

3.4.8 The Prg menu

The **Prg** menu contains commands that are used to write **giac** programs. The first entry, **Prg►New program** (equivalent to **Alt+P**), inserts a program level and brings up the program editor (see Section 12.1.1 p.835). The other entries are useful commands for writing **giac** programs.

3.4.9 The Graphic menu

The **Graphic** menu contains commands that are used to create graphs. The first entry, **Graphic►Attributs** (equivalent to **Alt+K**), brings up a window containing different attributes of the graph (such as line width, color, etc.). The other entries are commands for creating and manipulating graphs.

3.4.10 The Geo menu

The **Geo** menu contains commands that are used to work with two- and three-dimensional geometric figures. The first two entries, **Geo►New figure 2d** (equivalent to **Alt+G**) and **Geo►New figure 3d** (equivalent to **Alt+H**) create levels for two- and three-dimensional figures, respectively. (See Section 8.2 p.662.) The other menu items are for working with the figures.

3.4.11 The Spreadsheet menu

The **Spreadsheet** menu contains commands that are used to work with spreadsheets. (See Section 4.5 p.92.) The first menu item, **Spreadsheet►New spreadsheet** (equivalent to **Alt+T**), brings up a window where you can set the size and other attributes of a spreadsheet, after which one will be created. The submenus contain commands for working with spreadsheets. Notice that the spreadsheet itself will have menus that are the same as these submenus.

3.4.12 The Phys menu

The **Phys** menu contains submenus with various categories of constants, as well as functions for converting units.

3.4.13 The Highschool menu

The **Highschool** menu contains computer algebra commands that are useful at different levels of highschool. There is also a **Program** submenu with some program control functions.

3.4.14 The Turtle menu

The **Turtle** menu contains the commands that are used to create and control a Turtle screen. The first menu item, **Turtle►New turtle**, creates a Turtle drawing screen. The other menu items contain commands for working with the screen.

3.5 Configuring Xcas

3.5.1 The number of significant digits: Digits DIGITS

By default, **Xcas** uses and displays 12 significant digits, but you can set the number of digits to other positive integers. If you set the number of significant digits to a number less than 14, then **Xcas** will use the computer's floating point hardware, and so calculations will be done to more significant

digits than you asked for, but only the number of digits that you asked for will be displayed. If you set the number of significant digits to 14 or higher, then both the computations and the display will use that number of digits.

You can set the number of significant digits for **Xcas** by using the CAS configuration screen (see Section 3.5.7 p.73). The number of significant digits is stored in the variable **DIGITS** or **Digits**, so you can also set it by giving the variable **DIGITS** a new value, as in **DIGITS:= 20**. The value will be stored in the configuration file (see Section 3.5.10 p.78), and so can also be set there.

3.5.2 The language mode: **xcas_mode**

Xcas has its own language which it uses by default, but you can have it use Python (with the option having the \wedge character represent either exponentiation or the *exclusive or* operator), the language used by **Maple**, **MuPAD** or the **TI89** calculator.

You can set which language **Xcas** uses in the CAS configuration screen (see Section 3.5.7 p.73). You can also set the language with the **xcas_mode** command.

- The **xcas_mode** command takes one argument: an integer: 0, 1, 2, 3, 256 or 512.
 - **xcas_mode(0)**
to use the **Xcas** language.
 - **xcas_mode(1)**
to use the **Maple** language.
 - **xcas_mode(2)**
to use the **MuPAD** language.
 - **xcas_mode(3)**
to use the **TI89** language.
 - **xcas_mode(256)**
to use the Python language with \wedge representing exponentiation.
 - **xcas_mode(512)**
to use the Python language with \wedge representing *exclusive or*.

The language you choose will be stored in the configuration file (see Section 3.5.10 p.78), and so can also be set there.

3.5.3 The units for angles: **angle_radian**

By default, **Xcas** assumes that any angles you use (for example, as the argument to a trigonometric function) are being measured in radians. If you want, you can have **Xcas** use degrees.

You can set which angle measure **Xcas** uses in the CAS configuration screen (see Section 3.5.7 p.73). Your choice will be stored in the variable **angle_radian**; this will be 1 if you measure your angles in radians and 0 if you measure your angles in degrees. You can also change which angle measure you use by setting the variable **angle_radian** to the appropriate value. The angle measure you want to use will be stored in the configuration file (see Section 3.5.10 p.78), and so can also be set there.

3.5.4 Exact or approximate values: `approx_mode`

Some numbers, such as π and $\sqrt{2}$, can't be written down exactly as decimal numbers. When computing with such numbers, by default **Xcas** leaves them in exact, symbolic form. If you want, you can have **Xcas** automatically give you decimal approximations for these numbers.

You can set whether or not **Xcas** gives you exact or approximate values by using the CAS configuration screen (see Section 3.5.7 p.73). Your choice will be stored in the variable `approx_mode`, where a value of 0 means that **Xcas** will give you exact answers when possible and a value of 1 means that **Xcas** will give you decimal approximations. Your choice will be stored in the configuration file (see section 3.5.10), and so can also be set there.

3.5.5 Complex numbers: `cfactor` `complex_mode`

When factoring polynomials (see Section 6.12.10 p.209), by default **Xcas** won't introduce complex numbers if they aren't already being used. For example,

```
factor(x^2 + 2)
```

simply returns

$$x^2 + 2$$

but if an expression already involves complex numbers then **Xcas** uses them;

```
factor(i*x^2 + 2*i)
```

will return

$$(x - i\sqrt{2})(ix - \sqrt{2})$$

Xcas can also find complex roots when complex numbers are not present; for example, the command `cfactor` (see Section 6.12.10 p.209) will factor over the complex numbers.

`cFactor` is a synonym for `cfactor`.

```
cfactor(x^2 + 2)
```

returns

$$(x + i\sqrt{2})(x - i\sqrt{2})$$

If you want **Xcas** to use complex numbers by default, you can turn on complex mode. In complex mode,

```
factor(x^2 + 2)
```

returns

$$(x + i\sqrt{2})(x - i\sqrt{2})$$

You can turn on complex mode from the CAS configuration screen (see Section 3.5.7 p.73). This mode is determined by the value of the variable `complex_mode`; if this is 1 then complex mode is on, if this is 0 then complex mode is off. This option will be stored in the configuration file (see Section 3.5.10 p.78), and so can also be set there.

3.5.6 Complex variables: `complex_variables`

By default, new variables are assumed to be real; functions which work with the real and imaginary parts of variables will assume that a variable is real. For example, `re` returns the real part of its argument and `im` returns the imaginary part (see Section 6.10.2 p.197), and so

`re(z)`

returns

z

and

`im(z)`

returns

0

If you want variables to be complex by default, you can have Xcas use complex variable mode. You can set this from the CAS configuration screen (see Section 3.5.7 p.73). Your choice will be stored in the variable `complex_variables`, where a value of 0 means that Xcas will assume that variables are real and a value of 1 means that Xcas will assume that variables are complex. Your choice will be stored in the configuration file (see Section 3.5.10 p.78), and so can also be set there.

3.5.7 Configuring the computations

You can configure how Xcas computes by using the menu item **Cfg►Cas configuration** or by clicking on the status line. This will open a window with the following options:

1. **Prog style** (default: Xcas)

This has a menu from which you can choose a different language to program in; you can choose from Xcas, Python `^==*` (Python syntax, except that `^` will be the exponentiation operator as in Xcas rather than the *exclusive or* operator as in Python), Python `^==xor` (Python syntax, where `^` is the *exclusive or* operator), Maple, Mupad and TI89/92.

2. **eval** (default: 25)

This has an input field where you can type in a positive integer specifying the maximum number of recursions allowed when evaluating expressions.

3. **prog** (default: 1)

This has an input field where you can type in a positive integer specifying the maximum number of recursions allowed when executing programs.

4. **recurs** (default: 100)

This has an input field where you can type in a positive integer specifying the maximum number of recursive calls.

5. **debug** (default: 0)
This has an input field where you can type in a 0 or 1. If this is 1, then **Xcas** will display intermediate information on the algorithms used by **giac**. If this is 0, then no such information is displayed.
6. **maxiter** (default: 20)
This has an input field where you can type in an integer specifying the maximum number of iterations to be used in Newton's method.
7. **Float format** (default: **standard**)
This has a menu from which you can choose how to display decimal numbers. Your choices will be:
 - **standard** In standard notation, a number will be written out completely without using exponentials; for example, 15000.12 will be displayed as 15000.12.
 - **scientific** In scientific notation, a number will be written as a number between 1 and 10 times a power of ten; for example, 15000.12 will be displayed as 1.500012000000e+04 (where the number after **e** indicates the power of 10).
 - **engineer** In engineering notation, a number will be written as a number between 1 and 1000 times a power of ten, where the power of 10 is a multiple of three. For example, 15000.12 will be displayed as 15.00012e3.
8. **Digits** (default: 12)
This has an input field where you can type in a positive integer which will indicate the number of significant digits that **Xcas** will use.
9. **epsilon** (default: 1e-12)
This has an input field where you can type in a floating point number which will be the value of epsilon used by **epsilon2zero**, which is a function that replaces numbers with absolute value less than epsilon by 0 (see Section 6.59.1 p.650).
10. **proba** (default: 1e-15)
This has an input field where you can type in a floating point number. If this number is greater than zero, then in some cases **giac** can use probabilistic algorithms and give a result with probability of being false less than this value. (One such example of a probabilistic algorithm that **giac** can use is the algorithm to compute the determinant of a large matrix with integer coefficients.)
11. **approx** (default: unchecked)
This has a checkbox. If the box is checked, then exact numbers such as $\sqrt{2}$ will be given a floating point approximation. If the box is unchecked, then exact values will be used when possible. (See Section 3.5.4 p.72.)
12. **autosimplify** (default: 1)
This has an input field where you can type in 0, 1 or 2. A value of

0 means no automatic simplification will be done, a value of 1 means grouped simplification will be automatic. A value of 2 means that all simplification will be automatic.

13. **threads** (default: 1)
This has an input field where you can enter a positive integer to indicate the number of threads (for a possible future threaded version).
14. **Integer basis** (default: 10)
This has a menu from which you can choose an integer base to work in; your choices will be 8, 10 and 16.
15. **radian** (default: **checked**)
This has a checkbox. If the box is checked, then angles will be measured in radians, otherwise they will be measured in degrees.
16. **Complex** (default: **unchecked**)
This has a checkbox. If this box is checked, then **giac** will work in complex mode, meaning, for example, that polynomials will be factored with complex numbers if necessary.
17. **Cmplx_var** (default: **unchecked**)
This has a checkbox. If this box is checked, then variables will by default be assumed to be complex. For example, the expression **re(z)** won't be simplified, it will return **re(z)**. If this box is unchecked, then variables by default will be assumed to be real, and so **re(z)** will be simplified to **z**.
18. **increasing power** (default: **unchecked**)
This has a checkbox. If this box is checked, then polynomials will be written out in increasing powers of the variable; otherwise they will be written in decreasing powers.
19. **All_trig_sol** (default: **unchecked**)
This has a checkbox. If this box is checked, then **Xcas** will give the complete solutions of trigonometric equations. For example, the solution of $\cos(x) = 0$ will be given as $[(2n_0\pi + \pi)/2]$, where n_0 can be any integer. If this box is unchecked, then only the primary solutions of trigonometric equations will be given. For example, the solutions of $\cos(x) = 0$ will be the pair $[-\pi/2, \pi/2]$.
20. **Sqrt** (default: **checked**)
This has a checkbox. If this box is checked, then the **factor** command will factor second degree polynomials, even when the roots are not in the field determined by the coefficients. For example, **factor(x^2 - 3)** will return $(x - \sqrt{3})(x + \sqrt{3})$. If this box is unchecked, then **factor(x^2 - 3)** will return $x^2 - 3$.

This page also has buttons for applying the settings, saving the settings for future sessions, canceling any new settings, and restoring the default settings.

3.5.8 Configuring the graphics

You can configure each graphics screen by clicking on the `cfg` button on the graphics screen's control panel to the right of the graph. You can also change the default graphical configuration using the menu item `Cfg►Graph configuration`. You will then be given a window in which you can change the following options:

- `X-` and `X+`
These determine the x values for which calculations will be done.
- `Y-` and `Y+`
These determine the y values for which calculations will be done.
- `Z-` and `Z+`
These determine the z values for which calculations will be done.
- `t-` and `t+`
These determine the t values for which calculations will be done; when plotting parametric curves, for example.
- `WX-` and `WX+`
These determine the range of x values for the viewing window.
- `WY-` and `WY+`
These determine the range of y values for the viewing window.
- `TX` and `TY`
These determine the tick ranges on the x - and y -axes.
- `class_min`
This determines the minimum size of a statistics class.
- `class_size`
This determines the default size of a statistics class.
- `autoscale`
When checked, the graphic will be autoscaled.
- `ortho`
When checked, all axes of the graphic will be scaled equally.
- `>W` and `W>`
These are convenient shortcuts to copy the `X-`, `X+`, `Y-` and `Y+` values to `WX-`, `WX+`, `WY-` and `WY+`, or the other way around.

Note that the viewing window is not the same as the calculation window; if the calculation window is larger than the visible window, then you can scroll to bring other parts of the calculation window into view.

This page also has buttons for applying the settings, saving the settings for future sessions, or canceling any new settings.

3.5.9 More configuration

You can configure other aspects of **Xcas** (besides the computational aspects and graphics) using the menu item **Cfg►General configuration**. You will then be given a window in which you can change the following options:

- **Font**
This lets you choose a session font, the same as choosing the menu item **Cfg►Session font**.
- **Level**
This determines what type of level should be open when you start a new session.
- **browser**
This determines what browser **Xcas** will use when it requires one, for example when displaying help. If this is empty, **Xcas** will use its built-in browser.
- **Auto HTML help**
If this box is checked, then whenever you choose a function from a menu, a help page for that function will appear in your browser. Regardless of whether this box is checked or not, the help page will also appear in your browser if you enter *?function* from a command box.
- **Auto index help** If this box is checked, then whenever you choose a command from a menu, the help index page for that function will appear. This is the same page you get when you choose the command from the help index. (See Section 3.3 p.61.)
- **Print format**
This determines the paper size for printing and saving files. There is also a button you can use to have the printing done in landscape mode; if this button is not checked, the printing will be done in portrait.
- **Disable Tool tips**
If this box is checked, **Xcas** will stop displaying tool tips (see Section 3.3 p.61).
- **rows and columns**
These determine the default number of rows and columns for the matrix editor and spreadsheet (see Section 4.5 p.92).
- **PS view**
This determines what program is used to preview Postscript files.
- **Step by step**
If this is checked, then **Xcas** will not save context information.
- **Proxy**
This sets a proxy server for updates.

3.5.10 The configuration file: `widget_size` `cas_setup` `xcas_mode` `xyztrange`

When you save changes to your configuration, they are stored in a configuration file, which will be `.xcasrc` in your home directory in Unix and `xcas.rc` in Windows. This file will have four functions – `widget_size`, `cas_setup`, `xcas_mode` and `xyztrange` – which determine the configuration and which are evaluated when `Xcas` starts.

The `widget_size` command sets properties of the opening `Xcas` window.

- `widget_size` takes between 1 and 12 arguments. The arguments (in order) are:
 - *Font size*. The first argument is a positive integer specifying the font size. Optionally, this can be a bracketed list whose first number indicates the font and the second the font size.
 - *Horizontal and vertical offset*. The second and third arguments are horizontal and vertical distances in pixels from the upper left hand corner of the screen. They specify where the upper left corner of the `Xcas` window is when it opens.
 - *Window size*. The fourth and fifth arguments specify the width and height in pixels of the `Xcas` window when it opens.
 - *Keyboard* (see Section 3.2 p.58, item 3.2). The sixth argument is either 0 or 1; a 1 indicates that the on-screen keyboard will be open when `Xcas` starts, a 0 indicates that the keyboard will be hidden.
 - *Open browser*. The seventh argument is either 0 or 1; a 1 indicates that the browser will be automatically opened to display help for the selected command in the menu or index, a 0 indicates that the browser will not be automatically opened.
 - *Message window* (see Section 3.2 p.58, item 3.2). The eighth argument is either 0 or 1; a 1 indicates that `Xcas` will open with the message window, a 0 indicates that `Xcas` will open without the message window.
 - The ninth argument is currently not used.
 - *Browser name*. The tenth argument is a string with the name of the browser to use to read the help pages. A value of "builtin" means that `Xcas` will use a small browser built into `Xcas`.
 - *Starting level* (see Section 3.1 p.57). The eleventh argument indicates what level `Xcas` will start at; a 0 means command line, a 1 means program editor, a 2 means spreadsheet, and a 3 means a 2-d geometry screen.
 - *Postscript previewer*. The twelfth argument is a string with the name of a program for postscript previews; for example, "gv".

The `cas_setup` command determines how computations will be performed.

- `cas_setup` takes nine arguments. The arguments (in order) are:
 - *Approximate mode* (see Section 3.5.4 p.72). A 1 means *Xcas* works in approximate mode, a 0 means exact mode.
 - *Complex variables* (see Section 3.5.5 p.72). A 1 means *Xcas* works with complex variables, a 0 means real variables.
 - *Complex mode* (see Section 3.5.5 p.72). A 1 means *Xcas* works with in complex mode, a 0 means real mode.
 - *Radian* (see Section 3.5.3 p.71). A 1 means work in radians, a 0 means work in degrees.
 - *Display format* (see Section 3.5.7 p.73, item 7). A 0 means use the standard format to display numbers, a 1 means use scientific format, a 2 means use engineering format, and a 3 means use floating hexadecimal format (which is standardized with a non-zero first digit).
 - *Epsilon* (see Section 3.5.7 p.73, item 9). This is the value of `epsilon` used by *Xcas*.
 - *Digits*. This is the number of digits to use to display a float.
 - *Tasks*. This will be used in the future for parallelism.
 - *Increasing power*. This is 0 to display polynomials in increasing power, 1 to display polynomials in decreasing powers.

The `xcas_mode` command determines what computer language *Xcas* will use (see Section 3.5.2 p.71).

- The `xcas_mode` command takes one argument: an integer: 0, 1, 2, 3, 256 or 512.
 - `xcas_mode(0)`
to use the *Xcas* language.
 - `xcas_mode(1)`
to use the *Maple* language.
 - `xcas_mode(2)`
to use the *MuPAD* language.
 - `xcas_mode(3)`
to use the *TI89* language.
 - `xcas_mode(256)`
to use the *Python* language with `^` representing exponentiation.
 - `xcas_mode(512)`
to use the *Python* language with `^` representing *exclusive or*.

The `xyztrange` command sets or returns the values of the graphics configuration.

To set the values:

- `xyztrange` takes 12 arguments:

- $x-$ and $x+$, the beginning and the end of the x interval for which calculations will be done.
 - $y-$ and $y+$, the beginning and the end of the y interval for which calculations will be done.
 - $z-$ and $z+$, the beginning and the end of the z interval for which calculations will be done.
 - $t-$ and $t+$, the beginning and the end of the t interval for which calculations will be done, when plotting parametric curves, for example.
 - $wx-$ and $wx+$, the beginning and the end of the x values for the viewing window.
 - $wy-$ and $wy+$, the beginning and the end of the y values for the viewing window.
 - `show_axes`, to determine whether axes are shown or hidden (1 to show, 0 to hide).
 - `class_min`, the minimum size of a statistics class.
 - `class_size`, the default size of a statistics class.
- `xyztrange($x-,x+,y-,y+,z-,z+,t-,t+,wx-,wx+,wy-,wy+,show_axes,class_min,class_size$)` sets the parameters to the given values.

Note that the viewing window is not the same as the calculation window; if the calculation window is larger than the visible window, then you can scroll to bring other parts of the calculation window into view.

To return the values:

- `xyztrange` takes no arguments.
- `xyztrange()` returns a matrix where each row consists of a short description of the first twelve arguments along with their values.

3.6 Printing and saving

3.6.1 Saving a session

Each tab above the status line represents a session, the tab for the active session will be yellow. The label of each tab will be the name of the file that the session is saved in; if the session hasn't been saved the tab will read **Unnamed**.

You can save your current session by clicking on the **Save** button on the status line. If the session contains unsaved changes the **Save** button will be red; the button will be green when nothing needs to be saved. The first time that you save a session you will be prompted for a file name; you should choose a name that ends in **.xws**. Subsequent times that you save a session it will be saved in the same file; to save a session in a different file you can use the menu item **File►Save as**.

If you have a session saved in a file and you want to load it in a tab, you can use the menu item **File►Open**. From there you can choose a specific

file from a list or open a directory browser that you can use to choose a file. The directory browser can also be opened with **Alt-O**.

3.6.2 Saving a spreadsheet

If you have a spreadsheet in one of the levels, you can save it separately from the rest of the session.

When a spreadsheet is inserted it will have menus next to the level number. The **Table** menu has items that let you save the spreadsheet in different formats, as well as insert previously saved spreadsheets.

You can save a spreadsheet with the **Table►Save sheet as text** menu item. If you select that, you will be prompted for a file name; you should choose a file name that ends in **.tab**. Once you save a spreadsheet, there will be a button to the right of the menus which you can use to save any changes you make. If you want to save the spreadsheet under a different name, you can use the **Table►Save as alternate filename** menu entry.

You can save a spreadsheet in other formats. The **Table►Save as CSV** menu item will save a spreadsheet in a comma-separated values file, and the **Table►Save as mathml** menu item will save the spreadsheet in as a MathML file.

You can use the **Table** menu to insert previously saved spreadsheets; the menu item **Table►Insert** will bring up a directory browser that you can use to select a file to enter.

3.6.3 Saving a program

You can open up a program editor (see Section 12.1.1 p.835) with the menu item **Prg►New program** or with **Alt-P**. If you select this item, you will be prompted for information to fill out a template for a program and then be left in the program editor.

At the top of the program editor are menus and buttons, at the far right will be a **Save** button that you can press to save the program. The first time you save a program, you will be prompted for a file name; you should choose a name ending in **.cxx**. Once a program is saved, the file name will appear to the right of the **Save** button. If you want to save the program under a different name, you can use the **Prog►Save as** item from the program editor menu.

To insert a previously saved program, you can use the **Prog►Load** item from the program editor menu.

3.6.4 Printing a session

You can print a session with the **File►Print►To printer** menu item.

If you prefer to save the printed form as a file, you can use the **File►Print►Preview** menu item. You will be prompted for a file name to save the printed form in; the file will be a PostScript file, so the name should end in **.ps**. If you only want to save certain levels in printable form, you can use the **File►Print►Preview selected levels** menu item; this file will be encapsulated PostScript, so the name should end in **.eps**.

3.7 Translating to other computer languages

Xcas can translate a session, or parts of a session, to other computer languages; notably L^AT_EX and MathML.

3.7.1 Translating an expression to L^AT_EX: latex

The `latex` command translates expressions to L^AT_EX.

- `latex` takes one argument:
expr, an expression.
- `latex(expr)` returns the result of evaluating *expr* written in the L^AT_EX typesetting language.

Example.

Input:

```
latex(1+1/2)
```

Output:

```
\frac{3}{2}
```

3.7.2 Translating the entire session to L^AT_EX

To save your entire document as a complete L^AT_EX file, you can use the menu item **File►LaTeX►LaTeX preview**.

3.7.3 Translating graphical output to L^AT_EX: graph2tex graph3d2tex

You can see all of your graphic output at once on the **DispG** screen, which you can bring up with the command `DispG()`. (This screen can be cleared with the command line command `erase()`.) On the **DispG** screen there will be a **Print** menu; the **Print►LaTeX print** will give you several files `DispG.tex`, `DispG.dvi`, `DispG.ps` and `DispG.png` with the graphics in different formats. To save it without using the `DispG()` command you can use the `graph2tex` command.

The `graph2tex` command saves all current graphic output to a L^AT_EX file.

- `graph2tex` takes one argument:
filename.tex, the name of a file.
- `graph2tex("filename.tex")` saves all graphic output in L^AT_EX form to the file *filename.tex*.

Example.

Input:

```
graph2tex("myfile.tex")
```

results in a \LaTeX file named `myfile.tex` with the graphs. To save a three-dimensional graph, you can use the command `graph3d2tex`.

To save a single graph as a \LaTeX file, you can use the **M** menu to the right of the graph. Selecting **M►Export Print►Print (with LaTeX)** will save the current graph. You can also save a single graph by selecting that level, then use the menu item **File►LaTeX►LaTeX print selection**. This method will save the graph in several formats; `sessionname.tex`, `sessionname.dvi`, `sessionname.ps` and `sessionname.png`. If the session has not been saved and named, the files will begin with `sessionn` for some integer n .

3.7.4 Translating an expression to MathML: `mathml`

The `mathml` command translates expressions to MathML.

- `mathml` takes one argument:
expr, an expression.
- `mathml(expr)` returns the result of evaluating *expr* written in MathML.

Example.

Input:

```
mathml(1/4 + 1/4)
```

Output:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1 plus MathML 2.0//EN"
"http://www.w3.org/TR/MathML2/dtd/xhtml-math11-f.dtd" [
<!ENTITY mathml "http://www.w3.org/1998/Math/MathML">
]>
<html xmlns="http://www.w3.org/1999/xhtml">
<body>

<math mode="display" xmlns="http://www.w3.org/1998/Math/MathML">

<mfrac><mrow><mn>1</mn></mrow><mrow><mn>2</mn></mrow></mfrac>

</math><br/>

</body> </html>
```

which is the number $1/2$ in MathML form, along with enough information to make it a complete HTML document.

3.7.5 Translating a spreadsheet to MathML

You can translate an entire spreadsheet to MathML with the spreadsheet menu command **Table►Save as mathml**.

3.7.6 Indent an XML string: `xml_print`

The `xml_print` command formats an XML string.

- `xml_print` takes one argument:
str, a string, assumed to contain XML.
- `xml_print(str)` returns a string with the XML code indented for better readability. The default indentation is two spaces.

Example.

Input:

```
xml_print("<?xml version='1.0'?><root><child1>some
content</child1><child2></child2><child3/></root>")
```

Output:

```
<?xml version='1.0'?>
<root>
  <child1>some content</child1>
  <child2></child2>
  <child3/>
</root>
```

3.7.7 Export to presentation or content MathML: `export_mathml`

You can translate the result of an expression into various types of MathML with the `export_mathml` command.

- `export_mathml` takes one mandatory argument and one optional argument:
 - *expr*, an expression.
 - Optionally, *format*, which can be `content` or `display`, specifying what output format should be used.
- `export_mathml(expr [,format])` returns the result of evaluating *expr* written in MathML, with a single `math` block which will be a `semantics` block.
 - With no second argument, the `semantics` block will contain both presentation and content MathML.
 - With a second argument of `content`, the `semantics` block will only contain the content MathML.
 - With a second argument of `display`, the `semantics` block will only contain the presentation MathML.

Examples.

- *Input:*

```
xml_print(export_mathml(a+2*b))
```

Output:

```
<math xmlns='http://www.w3.org/1998/Math/MathML'>
  <semantics>
    <mrow xref='id5'>
      <mi xref='id1'>a</mi>
      <mo>+</mo>
      <mrow xref='id4'>
        <mn xref='id2'>2</mn>
        <mo>&it;</mo>
        <mi xref='id3'>b</mi>
      </mrow>
    </mrow>
    <annotation-xml encoding='MathML-Content'>
      <apply id='id5'>
        <plus/>
        <ci id='id1'>a</ci>
        <apply id='id4'>
          <times/>
          <cn id='id2' type='integer'>2</cn>
          <ci id='id3'>b</ci>
        </apply>
      </apply>
    </annotation-xml>
    <annotation encoding='giac'>a+2*b</annotation>
  </semantics>
</math>
```

- *Input:*

```
xml_print(export_mathml(a+2*b,content))
```

Output:

```
<math xmlns='http://www.w3.org/1998/Math/MathML'>
  <apply id='id5'>
    <plus/>
    <ci id='id1'>a</ci>
    <apply id='id4'>
      <times/>
      <cn id='id2' type='integer'>2</cn>
      <ci id='id3'>b</ci>
    </apply>
  </apply>
</math>
```

- *Input:*

```
xml_print(export_mathml(a+2*b,display))
```

Output:

```
<math xmlns='http://www.w3.org/1998/Math/MathML'>
  <mrow>
    <mi>a</mi>
    <mo>+</mo>
    <mrow>
      <mn>2</mn>
      <mo>&it;</mo>
      <mi>b</mi>
    </mrow>
  </mrow>
</math>
```

- *Input:*

```
s:=export_mathml(1/(x^2+1),display);
xml_print(s)
```

Output:

```
<math mode='display' xmlns='http://www.w3.org/1998/Math/MathML'>
  <mfrac>
    <mn>1</mn>
    <mrow>
      <msup>
        <mi>x</mi>
        <mn>2</mn>
      </msup>
      <mo>+</mo>
      <mn>1</mn>
    </mrow>
  </mfrac>
</math>
```

3.7.8 Translating a Maple file to Xcas: maple2xcas

The `maple2xcas` command translates a file of Maple commands to the Xcas language.

- `maple2xcas` takes two arguments:
 - *Maplefile*, the name of the Maple input file.
 - *XcasFile*, the file where you want to save the Xcas commands.
- `maple2xcas("MapleFile","XcasFile")` results in an Xcas file named *XcasFile* with the Maple commands in *MapleFile* translated to the Xcas language.

Chapter 4

Entry in Xcas

4.1 Suppressing output: `nodisp` ;;

If you enter a command into Xcas, the result will appear in the output box below the input. If you enter

```
a:= 2+2
```

then

4

will appear in the output box.

The `nodisp` command is used to evaluate an expression and suppress the output.

- `nodisp` takes one argument:
expr, an expression.
- `nodisp(expr)` evaluates *expr* but displays **Done** in place of the result.

Example.

Input:

```
nodisp(a:= 2+2)
```

Output:

Done

and **a** will be set to 4.

An alternate way of suppressing the output is to end the input with `;;`.

Example.

Input:

```
b:= 3+3;;
```

Output:

Done

and **b** will be set to 6.

4.2 Entering comments: `comment`

You can annotate an **Xcas** session by adding comments. You can enter a comment on the current line at any time by typing **Alt+C**. The line will appear in green text and conclude when you type **Enter**. Comments are not evaluated and so have no output. If you have started entering a command when you begin a comment, the command line with the start of the command will be pushed down so that you can finish it when you complete the comment.

You can open the browser using a comment line by entering the web address beginning with the `@` sign. If you enter the comment line

```
The Xcas homepage is at
@www-fourier.ujf-grenoble.fr/~parisse/giac.html
```

then the browser will open to the **Xcas** home page.

To add a comment to a program, rather than a session, you can use the `comment` command.

- `comment` takes one argument:
str, a string.
- `comment(str)` makes *str* a comment.

Alternatively, any part of a program between `//` and the end of the line is a comment. So both

```
bs() := {comment("Hello"); return "Hi there!";}
```

and

```
bs() := { // Hello
return "Hi there!";}
```

are programs with the comment "Hello".

4.3 Editing expressions

You can enter expressions on the command line, but **Xcas** also has a built-in expression editor that you can use to enter expressions in two dimensions, the way they normally look when typeset. When you have an expression in the editor, you can also manipulate subexpressions apart from the entire expression.

4.3.1 Entering expressions in the editor: an example

The expression

$$\frac{x+2}{x^2-4}$$

can be entered on the command line with

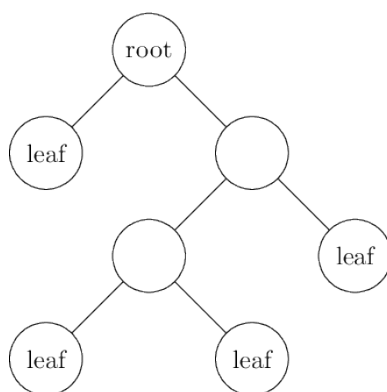
```
(x+2)/(x^2-4)
```


You also can use the expression editor to enter it visually, as $x + 2$ on top of $x^2 - 4$. To do this, you can start the expression editor with the **Alt+E** keystroke (or the **Expression ► New Expression** menu command). There will be a small M on the right side of the expression line, which is a menu with some commands you can use on the expressions. There will also be a 0 selected on the expression line and an on-screen keyboard at the bottom (see Section 3.2 p.58, item 3.2). If you type $x + 2$, it will overwrite the 0. To make this the top of the fraction, you can select it with the mouse (you can also make selections with the keyboard, as will be discussed later) and then type /. This will leave the $x + 2$ on the top of a horizontal fraction bar and the cursor on the bottom. To enter $x^2 - 4$ on the bottom, begin by typing x . Selecting this x and typing 2 will put on the superscript. Finally, selecting the x^2 and typing $- 4$ will finish the bottom. If you then hit **Enter**, the expression will be evaluated and will appear on the output line.

4.3.2 Subexpressions

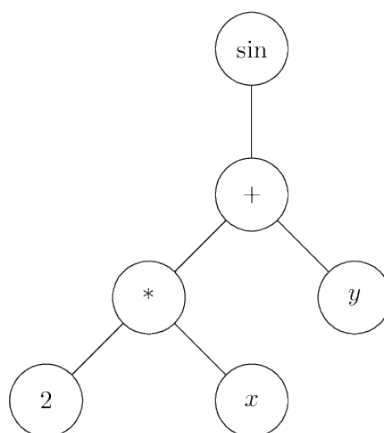
Xcas can operate on expressions in the expression editor or subexpressions of the expression. To understand subexpressions and how to select them, it helps to know that **Xcas** stores expressions as *trees*.

A tree, in this sense, consists of objects called nodes. A node can be connected to lower nodes, called the children of the node. Each node (except one) will be connected to exactly one node above it, called the parent node. One special node, called the root node, won't have a parent node. Two nodes with the same parent nodes are called siblings. Finally, if a node doesn't have any children, it is called a leaf. This terminology comes from a visual representation of a tree,



which looks like an upside-down tree; the root is at the top and the leaves are at the bottom.

Given an expression, the nodes of the corresponding tree are the functions, operators, variables and constants. The children of a function node are its arguments, the children of an operator node are its operands, and the constants and variables will be the leaves. For example, the tree for $\sin(2 * x + y)$ will look like

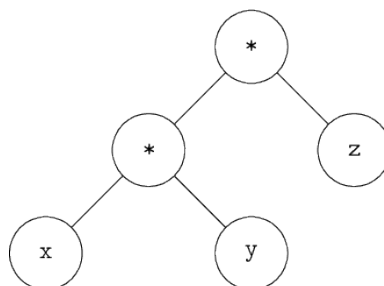


A subexpression of an expression will be a selected node together with the nodes below it. For example, both $2 * x$ and $2 * x + y$ are subexpressions of $\sin(2 * x + y)$, but $x + y$ is not.

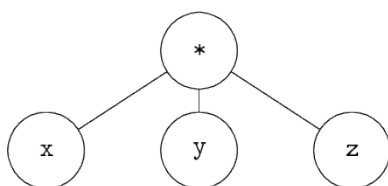
A subexpression of the contents of the expression editor can be selected with the mouse; the selection will appear white on a black background. A subexpression can also be chosen with the keyboard using the arrow keys. Given a selection:

- The up arrow will go to the parent node.
- The down arrow will go to the leftmost child node.
- The right and left arrows will go to the right and left sibling nodes.
- The control key with the right and left arrows will switch the selection with the corresponding sibling.
- If a constant or variable is selected, the backspace key will delete it. For other selections, backspace will delete the function or operator, and another backspace will delete the arguments or operands.

You can use the arrow keys to navigate the tree structure of an expression, which isn't always evident by looking at the expression itself. For example, suppose you enter $x*y*z$ in the editor. The two multiplications will be at different levels; the tree will look like



If you select the entire expression with the up arrow and then go to the M menu to the right of the line and choose eval, then the expression will look the same but, as you can check by navigating it with the arrow keys, the tree will look like



4.3.3 Manipulating subexpressions

If a subexpression is selected in the expression editor, then any menu command will be applied to that subexpression.

For example, suppose that you enter the expression

$$(x+1)*(x+2)*(x-1)$$

in the expression editor. Note that you can use the abilities of the editor to make this easier. First, enter `x+1`. Select this with the up arrow, then type `*` followed by `x+2`. Select the `x+2` with the up arrow and then type `*` followed by `x-1`. Using the up arrow again will select the `x-1`. Select the entire expression with the up arrow, and then select **eval** from the **M** menu. This will put all factors at the same level. Suppose you want the factors `(x+1)*(x+2)` to be expanded. You could select `(x+1)*(x+2)` with the mouse and do one of the following:

- Select the **Expression►Misc►normal** menu item. You will then have `normal((x+1)*(x+2))*(x-1)` in the editor. If you hit enter, the result $(x^2 + 3x + 2) * (x - 1)$ will appear in the output window.
- Select the **Expression►Misc►normal** menu item, so again you have `normal((x+1)*(x+2))*(x-1)` in the editor. Now if you select **eval** from the **M** menu, then the expression in the editor will become the result $(x^2 + 3x + 2) * (x - 1)$, which you can continue editing.
- Choose **normal** from the **M** menu. This will apply **normal** to the selection, and again you will have the result $(x^2 + 3x + 2) * (x - 1)$ in the editor.

There are also keystroke commands that you can use to operate on subexpressions that you've selected. There are the usual **Ctrl+Z** and **Ctrl+Y** for undoing and redoing. Some of the others are given in the following table.

Key	Action on selection
Ctrl+D	differentiate
Ctrl+F	factor
Ctrl+L	limit
Ctrl+N	normalize
Ctrl+P	partial fraction
Ctrl+R	integrate
Ctrl+S	simplify
Ctrl+T	copy L ^A T _E X version to clipboard

4.4 Previous results: `ans`

The `ans` command returns the results of previous commands.

- `ans` takes one optional argument:
Optionally, n , an integer (the number of the command beginning with 0).
- `ans($\langle n \rangle$)` returns the corresponding result; in particular, `ans(-1)` returns the previous result.

Example.

If the first command that you enter is:

Input:

2+5

resulting in

Output:

7

then later references to `ans(0)` will evaluate to 7.

Note that the argument to `ans` doesn't correspond to the line number in `Xcas`. For one thing, the line numbers begin at 1. What's more, if you go back and re-evaluate a previous line, then that will become part of the commands that `ans` keeps track of.

If you give `ans` a negative number, then it counts backwards from the current input. To get the latest output, for example, you can use `ans(-1)`. With no argument, `ans()` will also return the latest output.

Similarly, the `quest` command returns the previous inputs. Since these will often be simplified to be the same as the output, `quest(n)` sometimes has the same value as `ans(n)`.

You can also use `Ctrl` plus the arrow keys to scroll through previous inputs. With the cursor on the command line, `Ctrl+uparrow` will go backwards in the list of previous commands and display them on the current line, and `Ctrl+downarrow` will go forwards.

4.5 Spreadsheet

4.5.1 Opening a spreadsheet

You can open a spreadsheet (or a matrix editor) with the **Spreadsheet►New Spreadsheet** menu item or with the key `Alt+T`.

When you open a new spreadsheet, you will be given a configuration screen with the following options:

- **Variable** This has a input field where you can type in a variable name; the spreadsheet will be saved as a matrix in this variable.
- **Rows** and **Columns** These have input fields where you can type in positive integers specifying the number of rows and columns in the spreadsheet.

- **Eval** This has a checkbox. If the box is checked, then the spreadsheet will be re-evaluated every time you make a change to it. If it is not checked, it won't be re-evaluated when changes are made, but you can still re-evaluate the spreadsheet with the **eval** button on the spreadsheet menu bar.
- **Distribute** This has a checkbox. If it is checked, then entering a matrix will distribute the contents across an appropriate array of cells. If it is not checked, then the matrix will be put in one cell.
- **Landscape** This has a checkbox. If it is checked, then the graphical representation of the spreadsheet will be displayed below the spreadsheet. If it is not checked, then it will be displayed to the right of the spreadsheet.
- **Move right** This has a checkbox. If it is checked, then the cursor will move to the cell to the right of the current cell when data is entered. If this is not checked, the cursor will be moved to the cell below the current cell.
- **Spreadsheet** This has a checkbox. If it is checked, the spreadsheet will be formatted as a spreadsheet. If it is not checked, it will be formatted as a matrix.
- **Graph** This has a checkbox. If it is checked, the graphical representation of the spreadsheet will be displayed. If it is not checked, the graphical representation will not be displayed.
- **Undo history** This has an input field where you can type in a positive integer, specifying how many undo's can be performed at a time.

The configuration screen can be reopened with the **Edit►Configuration►Cfg window** menu attached to the spreadsheet.

4.5.2 The spreadsheet window

When you open a spreadsheet, the input line will become the spreadsheet.

	A	B	C	D	E
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	0
3	0	0	0	0	0
4	0	0	0	0	0
5	0	0	0	0	0
6	0	0	0	0	0
7	0	0	0	0	0
8	0	0	0	0	0
9	0	0	0	0	0
10	0	1	2	3	4

The top will be a menu bar with **Table**, **Edit** and **Maths** menus as well as **eval**, **val**, **init**, **2-d** and **3-d** buttons. To the right will be the name of the file the spreadsheet will be saved into. Below the menu bar will be two boxes; a box which displays the active cell (and can be used to choose a cell) and a command line to enter information into the cell. Below that will be a status line, you can click on this to return to the configuration screen.

Chapter 5

CAS building blocks

5.1 Numbers

Xcas works with both real and complex numbers. The real numbers can be integers, rational numbers, floating point numbers or symbolic constants.

You can enter an integer by simply typing the digits.

Input:

1234321

Output:

1234321

Alternatively, you can enter an integer in binary (base 2) by prefixing the digits (0 through 1) with **0b**, in octal (base 8) by prefixing the digits (0 through 7) with **0** or **0o**, and in hexadecimal (base 16) by prefixing the digits (0 through 9 and a through f) with **0x**. (See Section [6.4.1](#) p.130.)

Input :

0xab12

Output:

43794

You can enter a rational number as the ratio of two integers.

Input:

123/45

Output:

$$\frac{41}{15}$$

The result will be put in lowest terms. If the top is a multiple of the bottom, the result will be an integer.

Input:

123/3

Output:

41

A floating point number is regarded as an approximation to a real number. You can enter a floating point number by writing it out with a decimal point.

Input:

123.45

Output:

123.45

You can also enter a floating point number by entering a sequence of digits, with an optional decimal point, followed by **e** and then an integer, where the **e** represents “times 10 to the following power.”

Input:

1234e3

Output:

1234000.0

Floating point numbers with a large number of digits will be printed with **e** notation; you can control how other floats are displayed (see Section 3.5.7 p.73, item 7). An integer or rational number can be converted to a floating point number with **evalf** (see Section 6.8.1 p.171).

A complex number is a number of the form $a+bi$, where a and b are real numbers. The numbers a and b will be the same type of real number; one type will be converted to the other type if necessary (an integer can be converted to a rational number or a floating point number, and a rational number can be converted to a floating point number).

Input:

3 + 1.1i

Output:

3 + 1.1i

5.2 Symbolic constants: e pi infinity inf i euler_gamma

Xcas has the standard constants given by built-in symbols, given in the following table.

Symbol	Value
e (or %e)	the number $\exp(1)$
pi (or %pi)	the number π
infinity	unsigned ∞
+infinity (or inf)	$+\infty$
-infinity (or -inf)	$-\infty$
i (or %i)	the complex number i
euler_gamma	Euler's constant γ ; namely, $\lim_{n \rightarrow \infty} (\sum_{k=1}^n \frac{1}{k} - \ln(n))$

Since these numbers cannot be written exactly as standard decimal numbers, they are necessarily left unevaluated in exact results (see Section 3.5.4 p.72).

Input:

`2*pi`

Output:

2π

Input:

`2.0*pi`

Output:

6.28318530718

You can also use `evalf` (see Section 6.8.1 p.171), for example, to approximate one of the real-valued constants to as many decimal places as you want.

Input:

`evalf(pi,50)`

Output:

3.1415926535897932384626433832795028841971693993751

5.3 Sequences, sets and lists

5.3.1 Sequences: `seq[]` ()

A sequence is represented by a sequence of elements separated by commas, without delimiters or with either parentheses ((and)) or `seq[` and `]` as delimiters, as in:

Input:

`1,2,3,4`

or:

`(1,2,3,4)`

or:

`seq[1,2,3,4]`

Output:

1, 2, 3, 4

Note that the order of the elements of a sequence is significant. For example, if `B:=(5,6,3,4)` and `C:=(3,4,5,6)`, then `B==C` returns `false`. (A value can be assigned to a variable with the `:=` operator; see Section 5.4.1 p.100. Also, `==` is the test for equality; see Section 6.1.2 p.113.)

Note also that the expressions `seq[...]` and `seq(...)` are not the same (see Section 6.39.2 p.454 for information on `seq(...)`). For example, `seq([0,2])=(0,0)` and `seq([0,1,1,5])=[0,0,0,0,0]` but `seq[0,2]=(0,2)` and `seq[0,1,1,5]=(0,1,1,5)`

See Section 6.39 p.453 for operations on sequences.

5.3.2 Sets: `set[]`

To define a set of elements, put the elements separated by commas, with delimiters `%{` and `%}` or `set[` and `]`.

Input:

```
set[1,2,3,4]
```

or:

```
%{1,2,3,4%}
```

Output:

```
[[1,2,3,4]]
```

In the `Xcas` output, the set delimiters are displayed as `[[` and `]]` in order not to confuse sets with lists (see Section 5.3.3 p.98). For example, `[[1,2,3]]` is the set `%{1,2,3%}`, unlike `[1,2,3]` (normal brackets) which is the list `[1,2,3]`.

Input:

```
A:=%{1,2,3,4%}
```

or:

```
A:=set[1,2,3,4]
```

Output:

```
[[1,2,3,4]]
```

Input:

```
B:=%{5,5,6,3,4%}
```

or:

```
B:=set[5,5,6,3,4]
```

Output:

```
[[5,6,3,4]]
```

Remark.

The order in a set is not significant and the elements in a set are all distinct. If you input `B:=%{5,5,6,3,4%}` and `C:=%{3,4,5,3,6%}`, then `B==C` will return `true`.

See Section 6.41 p.489 for operations on sets.

5.3.3 Lists: `[]`

A list is delimited by `[` and `]`, its elements must be separated by commas. For example, `[1,2,5]` is a list of three integers. Lists are also called vectors in `Xcas`.

Lists can contain lists (for example, a matrix is a list of lists of the same size, see Section 6.44 p.502). Lists may be used to represent vectors (lists of coordinates), matrices, or univariate polynomials (lists of coefficients by decreasing order, see Section 6.27.1 p.351).

Lists are different from sequences, because sequences are flat: an element of a sequence cannot be a sequence. Lists are different from sets, because for a list, the order is important and the same element can be repeated in a list (unlike in a set where each element is unique). See Section 6.40 p.463 for operations on lists.

In Xcas output:

- list delimiters are displayed as `[,]`,
- matrix delimiters are displayed as `[,]`
- polynomial delimiters are displayed as `[[,]]`
- set delimiters are displayed as `[[,]]`.

5.3.4 Accessing elements

The elements of sequences and lists are indexed starting from 0 in Xcas syntax mode and from 1 in all other syntax modes (see Section 3.5.2 p.71). To access an element of a list or a sequence, follow the list with the index between square brackets.

Examples.

- *Input:*

$$L := [2, 5, 1, 4]$$

Output:

$$[2, 5, 1, 4]$$

- *Input:*

$$L[1]$$

Output:

$$5$$

- To access the last element of a list or sequence, you can put -1 between square brackets.

Input:

$$L[-1]$$

Output:

$$4$$

If you want the indices to start from 1 in Xcas syntax mode, you can enter the index between double brackets.

Example.

Input:

$$L[[1]]$$

Output:

$$2$$

5.4 Variables

5.4.1 Variable names

A variable or function name is a sequence of letters, numbers and under-scores that begins with a letter. If you define your own variable or function, you can't use the names of built-in variables or functions or other keywords reserved by `Xcas`.

5.4.2 Assigning values: `:=` `=>` = assign `sto` Store

You can assign a value to a variable with the `:=` operator. For example, to give the variable `a` the value of 4, you can enter

```
a:= 4
```

Alternatively, you can use the `=>` operator; when you use this operator, the value comes before the variable;

```
4 => a
```

The function `sto` (or `Store`) can also be used; again, the value comes before the variable (the value is stored into the variable);

```
sto(4,a)
```

After any one of these commands, whenever you use the variable `a` in an expression, it will be replaced by 4.

You can use sequences or lists to make multiple assignments at the same time. For example,

```
(a,b,c) := (1,2,3)
```

will assign `a` the value 1, `b` the value 2 and `c` the value 3. Note that this can be used to switch the values of two variables; with `a` and `b` as above, the command

```
(a,b) := (b,a)
```

will set `a` equal to `b`'s original value, namely 2, and will set `b` equal to `a`'s original value, namely 1.

Another way to assign values to variables, useful in Maple mode, is with the `assign` command. If you enter

```
assign(a,3)
```

or

```
assign(a = 3)
```

then `a` will have the value 3. You can assign multiple values at once; if you enter

```
assign([a = 1, b = 2])
```

then `a` will have the value 1 and `b` will have the value 2. This command can be useful in Maple mode, where solutions of equations are returned as equations. For example, if you enter (in Maple mode)

```
sol:= solve([x + y = 1, y = 2],[x,y])
```

(see Section 6.55.6 p.619) you will get

$$[x = -1, y = 2]$$

If you then enter

```
assign(sol)
```

the variable `x` will have value -1 and `y` will have the value 2. This same effect can be achieved in standard Xcas mode, where

```
sol:= solve([x + y = 1, y = 2],[x,y])
```

will return

$$[[-1, 2]]$$

In this case, the command

```
[x,y]:= sol[0]
```

will assign `x` the value -1 and `y` the value 2.

5.4.3 Assignment by reference: =<

A list is simply a sequence of values separated by commas and delimited by [and] (see Section 6.39 p.453). Suppose you give the variable `a` the value `[1,1,3,4,5]`,

```
a:= [1,1,3,4,5]
```

If you later assign to `a` the value `[1,2,3,4,5]`, then a new list is created. It may be better to just change the second value in the original list by reference. This can be done with the =< command. Recalling that lists are indexed beginning at 0, the command

```
a[1] =< 2
```

will simply change the value of the second element of the list instead of creating a new list, and is a more efficient way to change the value of `a` to `[1,2,3,4,5]`.

5.4.4 Copying lists: copy

If you enter

```
list1:= [1,2,3]
```

and then

```
list2:= list1
```

then `list1` and `list2` will be equal to the same list, not simply two lists with the same elements. In particular, if you change (by reference) the value of an element of `list1`, then the change will also be reflected in `list2`. For example, if you enter

```
list1[1] =< 5
```

then both `list1` and `list2` will be equal to `[1,5,3]`.

The `copy` command creates a copy of a list (or vector or matrix) which is equal to the original list, but distinct from it. For example, if you enter

```
list1:= [1,2,3]
```

and then

```
list2:= copy(list1)
```

then `list1` and `list2` will both be `[1,2,3]`, but now if you enter

```
list1[1] =< 5
```

then `list1` will be equal to `[1,5,3]` but `list2` will still be `[1,2,3]`.

5.4.5 Incrementing variables: `+=` `-=` `*=` `/=`

You can increase the value of a variable `a` by 4, for example, with

```
a:= a + 4
```

If beforehand `a` were equal to 4, it would now be equal to 8. A shorthand way of doing this is with the `+=` operator;

```
a += 4
```

will also increase the value of `a` by 4.

Similar shorthands exist for subtraction, multiplication and division. If `a` is equal to 8 and you enter

```
a -= 2
```

then `a` will be equal to 6. If you follow this with

```
a *= 3
```

then `a` will be equal to 18, and finally

```
a /= 9
```

will end with `a` equal to 2.

5.4.6 Storing and recalling variables and their values: `archive` `unarchive`

The `archive` command stores the values of variables for later use in a file of your choosing.

- `archive` takes two arguments:
 - *filename*, a filename in which to store values.
 - *vars*, a variable or list of variables.
- `archive("filename", vars)` saves the values of *vars* (or the values of the variables in the list) in file *filename*.

For example, if the variable `a` has the value 2 and the variable `bee` has the value `"letter"` (a string), then entering

```
archive("foo", [a, bee])
```

will create a file named `"foo"` which contains the values 2 and `"letter"` in a format meant to be efficiently read by `Xcas`.

The `unarchive` command will read the values from a file created with `archive`.

- `unarchive` takes one argument:
 - filename*, the filename.
- `unarchive("filename")` returns the value or list of values stored in *filename*.

Example.

With the file `"foo"` as above:

Input:

```
unarchive("foo")
```

Output:

```
[2, "letter"]
```

If you want to reassign these values to `a` and `bee`, you can enter

```
[a, bee] := unarchive("foo")
```

5.4.7 Copying variables: `CopyVar`

The `CopyVar` command copies the contents of one variable into another, without evaluating the contents.

- `CopyVar` takes two arguments:
 - *fromvar*, the name of a variable to copy from.
 - *tovar*, the name of a variable to copy to.
- `CopyVar(fromvar, tovar)` copies the unevaluated contents of *fromvar* into *tovar*.

Example.*Input:*

```
a:=c
c:=5
CopyVar(a,b)
```

*Output:**c**then:**b**Output:*

5

Changing the value of *c* will also change the output of *b*, since *b* contains *c*.

Input:

```
c:=10;;
b
```

Output:

10

5.4.8 Assumptions on variables: about additionally assume purge supposons and or

If **variable** is a purely symbolic variable (i.e., it doesn't have a value or any assumptions made about it), then

```
abs(variable)
```

will return

|variable|

since **Xcas** doesn't know what type of value the variable is supposed to represent.

The **assume** (or **supposons**) command lets you tell **Xcas** some properties of a variable without giving the variable a specific value. The **additionally** command can be used to add assumptions to a variable. The **about** command will display the current assumptions about a variable, and the **purge** command will remove all values and assumptions about a variable.

assume (or **supposons**) takes one mandatory argument and one optional argument:

- *assumptions*, statements about a variable (such as equalities and inequalities, possibly combined with **and** and **or**, and domains).
- Optionally, **additionally**, which indicates that the assumptions are to be added to previous assumptions, as opposed to replace them.

`assume(assumptions ⟨, additionally⟩)` places the assumptions on the variable. With no second argument, it will remove any previous assumptions.

- `additionally` takes one argument:
assumptions as above.
- `additionally(assumptions)` adds the assumptions to a variable without removing assumptions.
- `about` takes one argument:
var, the name of a variable.
- `about(var)` returns the current assumptions on the variable.
- `purge` takes one argument:
var, a variable name or a sequence of variable names.
- `purge(var)` removes any assumptions you have made about the variable *var* (or about all the variables in the sequence).

For example, if you enter

```
assume(variable > 0)
```

then Xcas will assume that `variable` is a positive real number, and so

```
abs(variable)
```

will be evaluated to

```
variable
```

You can put one or more conditions in the `assume` command by combining them with `and` and `or`. For example, if you want the variable `a` to be in $[2, 4) \cup (6, \infty)$, you can enter

```
assume((a >= 2 and a < 4) or a > 6)
```

If a variable has attached assumptions, then making another assumption with `assume` will remove the original assumptions. To add extra assumptions, you can either use the `additionally` command or give `assume` a second argument of `additionally`. If you assume that $b > 0$ with

```
assume(b > 0)
```

and you want to add the condition that $b < 1$, you can either enter

```
assume(b < 1, additionally)
```

or

```
additionally(b < 1)
```

As well as equalities and inequalities, you can make assumptions about the domain of a variable. If you want `n` to represent an integer, for example, you can enter

```
assume(n, integer)
```

If you want `n` to be a positive integer, you can add the condition

```
additionally(n > 0)
```

You can also assume a variable is in one of the domains `real`, `integer`, `complex` or `rational` (see Section 12.2.5 p.841).

You can check the assumptions on a variable with the `about` command. For the above positive integer `n`,

Input:

```
about(n)
```

Output:

```
assume[integer,[line[0,+infinity]], [0]]
```

The first element tells you that `n` is an integer, the second element tells you that `n` is between 0 and `+infinity`, and the third element tells you that the value 0 is excluded.

If you assume that a variable is equal to a specific value, such as

```
assume(c = 2)
```

then by default the variable `c` will remain unevaluated in later levels. If you want an expression involving `c` to be evaluated, you would need to put the expression inside the `evalf` command (see Section 6.8.1 p.171). After the above assumption on `c`, if you enter

```
evalf(c^2 + 3)
```

then you will get

7.0

Right below the `assume(c = 2)` command line there will be a slider; namely arrows pointing left and right with the value 2 between them. These can be used to change the values of `c`. If you click on the right arrow, the `assume(c = 2)` command will transform to

```
assume(c=[2.2, -10.0, 10.0, 0.0])
```

and the value between the arrows will be 2.2. Also, any later levels where the variable `c` is evaluated will be re-evaluated with the value of `c` now 2.2. The output to `evalf(c^2 + 3)` will become

7.84

The -10.0 and 10.0 in the `assume` line represent the smallest and largest values that `c` can become using the sliders. You can set them yourself in the `assume` command, as well as the increment that the value will change; if you want `c` to start with the value 5 and vary between 2 and 8 in increments of 0.05, then you can enter

```
assume(c = [5,2,8,0.05])
```

Recall the **purge** command removes assumptions about a variable.

Input:

```
purge(a)
```

then **a** will no longer have any assumptions made about it.

Input:

```
purge(a,b)
```

then **a** and **b** will no longer have any assumptions made about them.

5.4.9 Unassigning variables: VARS purge DelVar del restart rm_a_z rm_all_vars

Xcas has commands that help you keep track of what variables you are using and resetting them if desired. The **VARS** command will list all the variables that you are using, the **purge**, **DelVar** and **del** commands will delete selected variables, and the **rm_a_z** and **rm_all_vars** commands will remove classes of variables.

- **VARS** takes no arguments.
- **VARS()** returns a list of the variables that you have assigned values or made assumptions on.

Example.

Input:

```
a:= 1
anothervar:= 2
```

then:

```
VARS()
```

Output:

```
[a, anothervar]
```

The **purge** command will clear the values and assumptions you make on variables (see Section 5.4.8 p.104). For TI compatibility there is also **DelVar**, and for Python compatibility there is **del**.

- The **purge** command takes one argument:
var, the name of a variable.
- **purge(var)** clears the variable *var* of all values and assumptions.
- The **DelVar** (and **del**) commands take one argument:
var, the name of a variable.
- **Delvar var** (or **del var**) removes the values attached to *var*. (Note that they do not take their argument in parentheses.

Example.

To clear the variable **a**:

Input:

```
purge(a)
```

or (for TI compatibility):

Input:

```
DelVar a
```

or (for Python compatibility):

Input:

```
del a
```

The **rm_all_vars** and **restart** commands clear the values and assumptions you have made on all variables you can use.

- **rm_all_vars** takes no arguments.
- **rm_all_vars()** removes all the values that you have attached to variables.
- **restart** takes no arguments.
- **restart** removes all the values that you have attached to variables. (Note that it does not use parentheses.)

The **rm_a_z** command clears the values and assumptions on all variables with single lowercase letter names.

- **rm_a_z** takes no arguments.
- **rm_a_z()** purges all variables whose names are one letter and lowercase.

Example.

If you have variables names **A,B,a,b,myvar**, then after:

Input:

```
rm_a_z()
```

you will only have the variables named **A,B,myvar**.

5.4.10 The CST variable

The menu available with the **cust** button in the bandeau on the onscreen keyboard (see Section 3.2 p.58, item 3.2) is defined with the **CST** variable. It is a list where each list item determines a menu item; a list item is either a builtin command name or a list itself consisting of a string to be displayed in the menu and the input to be entered when the item is selected.

For example, to create a custom defined menu with the builtin function **diff**, a user defined function **foo**, and a menu item to insert the number $22/7$, you can:

Input:

```
CST:= [diff,["foo",foo],["My pi approx",22/7]]
```

Note that if the input to be entered is a variable and the variable has a value when CST is defined, then CST will contain the value of the variable. For example,

Input:

```
app:= 22/7
CST:= [diff,["foo",foo],["My pi approx",app]]
```

will be equivalent to the previous definition of CST. However, if the variable does not have a value when CST is defined, for example:

Input:

```
CST:= [diff,["foo",foo],["My pi approx",app]]
app:= 22/7
```

then CST will behave as the previous values to begin with, but in this case if the variable `app` is changed, the the result of pressing the `My pi approx` button will change also.

Since CST is a list, a function can be added to the `cust` menu with the `concat` command (see Section 6.40.13 p.472);

Input:

```
CST:= concat(CST,evalc)
```

will add the `evalc` command to the `cust` menu.

5.5 Functions

5.5.1 Defining functions

Similar to how you can assign a value to a variable (see Section 5.4.2 p.100), you can use the `:=` and `=>` operators to define a function; both

```
f(x):= x^2
```

and

```
x^2 => f(x)
```

give the name `f` to the function which takes a value and returns the square of the value. In either case, if you then enter:

Input:

```
f(3)
```

you will get:

Output:

9

You can define an anonymous function, namely a function without a name, with the `->` operator; the squaring function can be written

```
x -> x^2
```

You can use this form of the function to assign it a name; both

```
f := x -> x^2
```

and

```
x -> x^2 => f
```

are alternate ways to define **f** as the squaring function.

You can similarly define functions of more than one variable. For example, to define a function which takes the lengths of the two legs of a right triangle and returns the hypotenuse, you could enter

```
hypot(a,b) := sqrt(a^2 + b^2)
```

or

```
hypot := (a,b) -> sqrt(a^2 + b^2)
```

5.6 Directories

5.6.1 Working directories: `pwd` `cd`

Xcas has a working directory where it stores files that it creates; typically this is the user's home directory. The `pwd` command will tell you what the current working directory is, and the `cd` command lets you change it.

- `pwd` takes no arguments.
- `pwd()` returns the name of the current working directory.

Example.

Input:

```
pwd()
```

Output: might be something like:

```
/home/username
```

- The `cd` command takes one argument:
dirname, the name of a directory (a string).
- `cd(dirname)` changes the working directory to *dirname*.

Example.

If you enter:

Input:

```
cd("foo")
```

or (on a Unix system):

Input:

```
cd("/home/username/foo")
```

then the working directory will change to the directory `foo`, if it exists. Afterwards, any files that you save from `Xcas` will be in that directory.

To load or read a file, it will need to be in the working directory. Note that if you have the same file name in different directories, then loading the file name will load the file in the current directory.

5.6.2 Reading files: read load

Information for `Xcas` can be stored in a file; this information can be read with the `read` or `load` command, depending on the type of information.

The `read` command reads a file containing `Xcas` information, such as a program that you saved (see Section 3.6.3 p.81) or simply commands that you typed into a file with a text editor. The file should have the suffix `.cxx`.

- `read` takes one argument:
filename, the name of a file (a string) containing a saved program (see Section 3.6.3 p.81) or other commands.
- `read(filename)` reads the content of the file.

Example.

If you have a file named `myfunction.cxx`,

Input:

```
read("myfunction.cxx")
```

will read in the file, as long as the directory is in the current working directory. If the file is in a different directory, you can still read it by giving the path to the file,

Input:

```
read("/path/to/file/myfunction.cxx")
```

The `load` command reads in a saved session (see Section 3.6.1 p.80), which will end in `.xws`.

- `load` takes one argument:
filename, the name of a file (a string) containing a saved session.
- `load(filename)` loads the session stored in *filename*.

Example.

If you have a session saved in the file `mysession.xws`,

Input:

```
load("mysession.xws")
```

loads `mysession.xws`.

5.6.3 Internal directories: NewFold SetFold GetFold DelFold VARS

You can create a directory that isn't actually on your hard drive but is treated like one by `Xcas` with the command `NewFold`.

- `NewFold` takes one argument: *MyIntDir*, a variable name (see Section 5.4.1 p.100).
- `NewFold(MyIntDir)` creates a new internal directory named *MyIntDir*. (Note that quotation marks are not used.)

Internal directories will be listed with the `VARS()` command (see Section 5.4.9 p.107).

To actually use this directory, you'll have to use the `SetFold` command.

- The `SetFold` command takes one argument:
MyIntDir, the variable name of an internal directory created with `NewFold`.
- `SetFold(MyIntDir)` makes *MyIntDir* the working directory (see Section 5.6.1 p.110).

Finally, you can print out the internal directory that you are in with the `GetFold` command.

- `GetFold` takes no arguments.
- `GetFold()` returns the name of the current internal directory.

Example.

Input:

```
GetFold()
```

will display the current internal directory.

The `DelFold` command will delete an internal directory.

- `DelFold` takes one argument:
MyIntDir, the variable name of an internal directory.
- `DelFold(MyIntDir)` will delete the directory if it is empty.

Chapter 6

The CAS functions

6.1 Booleans

6.1.1 Boolean values: true false

The symbols `true` and `false` are *booleans*, and are meant to indicate a statement is true or false.

These constants have synonyms:

- `true` is the same as `TRUE` or `1`.
- `false` is the same as `FALSE` or `0`.

A function which returns a boolean is called a *test* (or a *condition* or a *boolean function*).

6.1.2 Tests: == != > >= < <=

The usual comparison operators between numbers are examples of tests. In `Xcas`, they are the infix operators:

`==`

`a==b` tests the equality between *a* and *b* and returns `1` if *a* is equal to *b* and `0` otherwise.

Look out !

Note that `a=b` is **not** a boolean!!!! This form is used to state that the expression *is* an equality, perhaps with the intent to solve it. To *test* for equality, you need to use `a==b`, which *is* a boolean.

`!=`

`a!=b` returns `1` if *a* and *b* are different and `0` otherwise.

`>=`

`a>=b` returns `1` if *a* is greater than or equal to *b* and `0` otherwise.

`>`

`a>b` returns `1` if *a* is strictly greater than *b* and `0` otherwise.

`<=`

`a<=b` returns `1` if *a* is less than or equal to *b* and `0` otherwise.

<

 $a < b$ returns 1 if a is strictly less than b and 0 otherwise.

6.1.3 Defining functions with boolean tests: ifte ?: when

You can use boolean tests to define functions not given by a single simple formula. Notably, you can use the `ifte` command or `?:` operator to define piecewise-defined functions.

- `ifte` takes three arguments:
 - *condition*, a boolean condition.
 - *true-result*, the result to return if *condition* is true.
 - *false-result*, the result to return if *condition* is false.
- `ifte(condition, true-result, false-result)` returns *true-result* if *condition* is true and returns *false-result* if *condition* is false.

Example.

You can define your own absolute value function with:

Input:

```
myabs(x) := ifte(x >= 0, x, -1*x)
```

Afterwards, entering:

Input:

```
myabs(-4)
```

will return:

```
4
```

However, `myabs` will return an error if it can't evaluate the condition.

Input:

```
myabs(x)
```

Output:

```
Ifte: Unable to check test Error: Bad Argument Value
```

The `?:` construct behaves similarly to `ifte`, but is structured differently and doesn't return an error if the condition can't be evaluated.

- The `?:` construct takes three arguments:
 - *condition*, a boolean condition.
 - *true-result*, the result to return if *condition* is true.
 - *false-result*, the result to return if *condition* is false.
- `condition?true-result:false-result` returns *true-result* if *condition* is true and returns *false-result* if *condition* is false.

Example.

You can define your absolute value function with

$$\text{myabs}(x) := (x \geq 0)? \quad x: \quad -1*x$$

If you enter

$$\text{myabs}(-4)$$

you will again get

$$4$$

but now if the conditional can't be evaluated, you won't get an error.

Input:

$$\text{myabs}(x)$$

Output:

$$((x \geq 0)? \quad x: \quad -x)$$

The **when** and **IFTE** commands are prefixed synonyms for the **?:** construct.

- **when** (and **IFTE**) take three arguments:
 - *condition*, a boolean condition.
 - *true-result*, the result to return if *condition* is true.
 - *false-result*, the result to return if *condition* is false.
- **when**(*condition*, *true-result*, *false-result*) (and **IFTE**(*condition*, *true-result*, *false-result*)) return *true-result* if *condition* is true and returns *false-result* if *condition* is false.

$$(condition)? \quad true-result: \quad false-result$$

$$\text{when}(condition, \quad true-result, \quad false-result)$$

and

$$\text{IFTE}(condition, \quad true-result, \quad false-result)$$

all represent the same expression.

If you want to define a function with several pieces, it may be simpler to use the **piecewise** function.

- **piecewise** takes an unspecified (odd) number of arguments:
 - *cond*₁, *return*₁, *cond*₂, *return*₂, . . . , *cond*_n, *return*_n, an arbitrary number of pairs of conditions and corresponding return values.
 - *default*, a result to return if none of the conditions are true.

- `piecewise(cond1, return1, . . . , condn, returnn, default)` returns `returnk` if `condk` is the first true condition, or `default` if none of the conditions are true.

Example.

To define

$$f(x) = \begin{cases} -2 & \text{if } x < -2 \\ 3x + 4 & \text{if } -2 \leq x < -1 \\ 1 & \text{if } -1 \leq x < 0 \\ x + 1 & \text{if } x \geq 0 \end{cases}$$

you can enter:

Input:

```
f(x):= piecewise(x < -2, -2, x < -1, 3*x+4, x < 0, 1, x + 1)
```

6.1.4 Boolean operators: or xor and not

Booleans can be combined to form new booleans. For example, with **and**: the statement “*boolean 1 and boolean 2*” is true if both *boolean 1* and *boolean 2* are true, otherwise the statement is false.

Xcas has the standard boolean operators, as follows (*a* and *b* are two booleans):

or (**or ||**)

These are infix operators. (*a or b*) (or (*a || b*)) returns 0 (or **false**) if *a* and *b* are both equal to 0 (or **false**) and returns 1 (or **true**) otherwise.

xor

This is an infix operator. It is the “exclusive or” operator, meaning “one or the other but not both”. (*a xor b*) returns 1 if *a* is equal to 1 and *b* is equal to 0 or if *a* is equal to 0 and *b* is equal to 1, and returns 0 if *a* and *b* are both equal to 0 or if *a* and *b* are both equal to 1.

and (**or &&**)

These are infix operators. (*a and b*) (or (*a && b*)) returns 1 (or **true**) if *a* and *b* are both equal to 1 (or **true**) and returns 0 (or **false**) otherwise.

not

This is a prefixed operator. **not**(*a*) returns 1 (or **true**) if *a* is equal to 0 (or **false**), and 0 (or **false**) if *a* is equal to 1 (or **true**).

Examples.

- *Input:*

```
1>=0 or 1<0
```

Output:

- *Input:*

$$1 \geq 0 \text{ xor } 1 > 0$$

Output:

$$0$$

- *Input:*

$$1 \geq 0 \text{ and } 1 > 0$$

Output:

$$1$$

- *Input:*

$$\text{not}(0 == 0)$$

Output:

$$0$$

6.1.5 Transforming a boolean expression to a list: `exp2list`

The `exp2list` command can transform certain booleans into a list.

- `exp2list` takes one argument: *eqseq*, a sequence of equalities (or inequalities) connected with `ors`, such as $(x = a_1) \text{ or } \dots \text{or } (x = a_n)$.
- `exp2list(eqseq)` returns the list $[a_1, \dots, a_n]$ of right-hand sides of the (in)equalities.

The `exp2list` command is useful in TI mode for easier processing of the answer to a `solve` command.

Examples.

- *Input:*

$$\text{exp2list}((x=2) \text{ or } (x=0))$$

Output:

$$[2, 0]$$

- *Input:*

$$\text{exp2list}((x>0) \text{ or } (x<2))$$

Output:

$$[0, 2]$$

- In TI mode

Input:

$$\text{exp2list}(\text{solve}((x-1)*(x-2)))$$

Output:

$$[1, 2]$$

6.1.6 Transforming a list into a boolean expression: `list2exp`

The `list2exp` command is the inverse of `exp2list`; it takes lists and transforms them into boolean expressions. It can do this in two ways.

The first way:

- `list2exp` takes two arguments:
 - L , a list of values of the form $[a_1, \dots, a_n]$
 - x , a variable name.
- `list2exp(L, x)` returns the boolean expression $((x = a_1) \text{ or } \dots (x = a_n))$.

Examples.

- *Input:*

```
list2exp([0,1,2],a)
```

Output:

$$a = 0 \vee a = 1 \vee a = 2$$

- *Input:*

```
list2exp(solve(x^2-1=0,x),x)
```

Output:

$$x = -1 \vee x = 1$$

Alternatively:

- `list2exp` takes two arguments:
 - L , a list where each element of L is itself a list of n values of the form $[a_1, \dots, a_n]$.
 - $vars$, a list $[x_1, \dots, x_n]$ of n variable names.

In this case:

- `list2exp($L, vars$)` returns a boolean expression of the form $((x_1 = a_1) \text{ and } \dots \text{ and } (x_n = a_n))$ for each list of n values in the first argument, combined with `ors`.

Example.

Input:

```
list2exp([[3,9], [-1,1]], [x, y])
```

Output:

$$x = 3 \wedge y = 9 \vee x = -1 \wedge y = 1$$

6.1.7 Evaluating booleans: `evalb`

The Maple command `evalb` evaluates a boolean expression (see Section 6.1 p.113). Since `Xcas` evaluates booleans automatically, it includes a `evalb` command only here for compatibility and is equivalent to `eval` (see Section 6.12.1 p.203).

- `evalb` takes one argument:
bool, a boolean expression.
- `evalb(bool)` returns 1 if *bool* is true and returns 0 otherwise.

Examples.

- *Input:*

```
evalb(sqrt(2)>1.41)
```

or:

```
sqrt(2)>1.41
```

Output:

```
1
```

- *Input:*

```
evalb(sqrt(2)>1.42)
```

or:

```
sqrt(2)>1.42
```

Output:

```
0
```

6.2 Bitwise operators

6.2.1 Basic operators: `bitor` `bitxor` `bitand`

Bitwise operators operate on the base 2 representations of integers, even if they are not presented in base 2. For example, the bitwise `or` (see Section 6.1.4 p.116) operator will take two integers and return an integer whose base 2 digits are the logical *ors* of the corresponding base two digits of the inputs (see Section 6.1.4 p.116). Thus, to find the bitwise *or* of 6 and 4, look at their base 2 representations, which are `0b110` (the `0b` prefix indicates that it's in base 2, see Section 5.1 p.95) and `0b100`, respectively. The logical *or* of their rightmost digits is `0 or 0=0`. The logical *or* of their next digits is `1 or 0=1`, and the logical *or* of their remaining digits is `1 or 1=1`. So the bitwise *or* of 6 and 4 is `0b110`, which is 6.

To work with bitwise operators, it isn't necessary but it may be useful to work with integers in a base which is a power of 2. The integers can be entered in binary (base 2), octal (base 8) or hexadecimal (base 16) (see Section 6.4.1 p.130). To write an integer in binary, prefix it with `0b`; to write an integer in octal, prefix it with `0` or `0o`; and to write a integer in hexadecimal (base 16), prefix it with `0x`. Integers may also be output in octal or hexadecimal notation (see Section 3.5.7 p.73, item 14).

There are bitwise versions of the logical operators `or`, `xor` and `and`; they are all prefixed operators which take two arguments, which are both integers.

- `bitor` is bitwise logical inclusive `or`.

Input:

```
bitor(0x12,0x38)
```

or:

```
bitor(18,56)
```

Output:

```
58
```

because:

18 is written `0x12` in base 16 or `0b010010` in base 2,
 56 is written `0x38` in base 16 or `0b111000` in base 2,
 hence `bitor(18,56)` is `0b111010` in base 2 and so is equal to 58.

- `bitxor` is bitwise logical exclusive `or`.

Input:

```
bitxor(0x12,0x38)
```

or:

```
bitxor(18,56)
```

Output:

```
42
```

because:

18 is written `0x12` in base 16 and `0b010010` in base 2,
 56 is written `0x38` in base 16 and `0b111000` in base 2,
`bitxor(18,56)` is written `0b101010` in base 2 and so, is equal to 42.

- `bitand` is bitwise logical `and`.

Input:

```
bitand(0x12,0x38)
```


or:

```
bitand(18,56)
```

Output:

16

because:

18 is written 0x12 in base 16 and 0b010010 in base 2,

56 is written 0x38 in base 16 and 0b111000 in base 2,

bitand(18,56) is written 0b010000 in base 2 and so is equal to 16.

6.2.2 Bitwise Hamming distance: hamdist

The Hamming distance between two integers is the number of differences between the bits of the two integers. The `hamdist` operator finds the Hamming distance between two integers.

- `hamdist` takes two arguments:
 m and n , both integers.
- `hamdist(m , n)` returns the Hamming distance between m and n .

Example.

Input:

```
hamdist(0x12,0x38)
```

or:

```
hamdist(18,56)
```

Output:

3

because:

18 is written 0x12 in base 16 and 0b010010 in base 2,

56 is written 0x38 in base 16 and 0b111000 in base 2,

hamdist(18,56) is equal to 1+0+1+0+1+0 and so is equal to 3.

6.3 Strings

6.3.1 Characters and strings: "

Strings are delimited with quotation marks, ". A character is a string of length one.

Do not confuse " with ' (or `quote`) which is used to prevent evaluation of an expression (see Section 6.12.4 p.205). For example, "a" returns a string with one character but 'a' or `quote(a)` returns the variable `a` unevaluated.

When a string is entered on a command line, it is evaluated to itself, hence the output is the same string. You can use + to concatenate two strings or a string and another object (where the other object will be converted to a string, see Section 6.3.12 p.128).

Examples.

- *Input:*

```
"Hello"
```

Output:

```
"Hello"
```

- *Input:*

```
"Hello"+" , how are you?"
```

Output:

```
"Hello, how are you?"
```

- *Input:*

```
"Hello"+ 123
```

Output:

```
"Hello123"
```

You can refer to a particular character of a string using index notation, like for lists (see Section 6.40 p.463). Indices begin at 0 in Xcas mode, 1 in other modes.

Example.

Input:

```
"Hello"[1]
```

Output:

```
"e"
```

6.3.2 The newline character: \n

A newline can be inserted into a string with \n.

Example.

Input:

```
Hello\nHow are you?
```

Output:

```
Hello
How are you?
```

6.3.3 The length of a string: `size` `length`

The `size` command can find the length of a string (as well as the length of lists in general, see Section 6.39.3 p.457).

`length` is a synonym for `size`.

- `size` takes one argument:
 - *str*, a string.
- `size(str)` returns the length of the string.

Example.

Input:

```
size("hello")
```

Output:

```
5
```

6.3.4 The left and right parts of a string: `left` `right`

The `left` and `right` commands can find the left and right parts of a string. (See Section 6.15.3 p.236, Section 6.37.1 p.445, Section 6.38.2 p.448, Section 6.40.6 p.467, Section 6.55.4 p.618 and Section 6.55.5 p.618 for other uses of `left` and `right`.)

- `left` takes two arguments:
 - *str*, a string.
 - *n*, a non-negative integer.
- `left(str,n)` returns the first *n* characters of the string *str*.

Example.

Input:

```
left("hello",3)
```

Output:

```
"hel"
```

- `right` takes two arguments:
 - *str*, a string.
 - *n*, a non-negative integer.
- `right(str,n)` returns the last *n* characters of the string *str*.

Example.

Input:

```
right("hello",4)
```

Output:

```
"ello"
```

6.3.5 First character, middle and end of a string: `head` `mid` `tail`

The `head` command finds the first character of a string.

- `head` takes one argument:
str, a string.
- `head(str)` returns the first character of the string *str*.

Example.

Input:

```
head("Hello")
```

Output:

```
"H"
```

The `mid` command finds a selected part from the middle of a string.

- `mid` takes three arguments:
 - *str*, a string.
 - *p*, an integer for the starting index of the result.
 - *q*, an integer *q* for the length of the string.
- `mid(str, p, q)` returns the part of the string *str* starting with the character at index *p* with length *q*. (Remember that the first index is 0 in Xcas mode.)

Example.

Input:

```
mid("Hello",1,3)
```

Output:

```
"ell"
```

The `tail` command removes the first character of a string.

- `tail` takes one argument:
str, a string.
- `tail(str)` returns the string *str* without its first character.

Input:

```
tail("Hello")
```

Output:

```
"ello"
```

6.3.6 Concatenation of a sequence of words: `cumSum`

The `cumSum` command works on strings like it does on expressions by doing partial concatenation (see Section 6.40.26 p.481).

- `cumSum` takes one argument:
L, a list of strings.
- `cumSum(L)` returns a list of strings where the element of index *k* is the concatenation of the strings in *L* with indices 0 to *k*.

Example.

Input:

```
cumSum("Hello, ","is ","that ","you?")
```

Output:

```
"Hello, ","Hello, is ","Hello, is that ","Hello, is that you?"
```

6.3.7 ASCII code of a character: `ord`

The `ord` command finds the ASCII code of a character.

- `ord` takes one argument:
str, a string (or a list of strings). `ord(str)` returns the ASCII code of the first character of *str* (or the list of the ASCII codes of the first characters of the elements of the list *str*).

Example.

Input:

```
ord("a")
```

Output:

```
97
```

Input:

```
ord("abcd")
```

Output:

```
97
```

Input:

```
ord(["abcd","cde"])
```

Output:

```
[97, 99]
```

Input:

```
ord(["a","b","c","d"])
```

Output:

```
[97, 98, 99, 100]
```

6.3.8 ASCII code of a string: `asc`

The `asc` command finds the ASCII codes of all the characters in a string.

- `asc` takes one argument:
str, a string.
- `asc(str)` returns the list of the ASCII codes of the characters of *s*.

Examples.

- *Input:*

```
asc("abcd")
```

Output:

```
[97, 98, 99, 100]
```

- *Input:*

```
asc("a")
```

Output:

```
[97]
```

6.3.9 String defined by the ASCII codes of its characters: `char`

The `char` command translates ASCII codes to strings.

- `char` takes one argument:
c, an integer representing an ASCII code or a list of ASCII codes.
- `char(c)` returns the string whose character has ASCII code *c* or whose characters have ASCII codes the elements of the list *c*.

Example.

Input:

```
char([97, 98, 99, 100])
```

Output:

```
"abcd"
```

Input:

```
char(97)
```

Output:

```
"a"
```

Note that there are 256 ASCII codes, 0 through 255. If `asc` is given an integer *c* not in that range, it will use the integer in that range which equals *c* modulo 256.

Input:

```
char(353)
```

Output:

```
"a"
```

because $353 - 256 = 97$.

6.3.10 Finding a character in a string: `inString`

The `inString` command tests to see if a string contains a character.

- `inString` takes two arguments:
 - *str*, a string.
 - *c*, a character.
- `inString(str,c)` returns the index of its first occurrence of the character *c* in the string *str*, or -1 if *c* does not occur in *str*.

Examples.

- *Input:*

```
inString("abcded","d")
```

Output:

```
3
```

- *Input:*

```
inString("abcd","e")
```

Output:

```
-1
```

6.3.11 Concatenating objects into a string: `cat`

The `cat` command transforms a sequence of objects into a string.

- `cat` takes one argument:
 - seq*, a sequence of objects.
- `cat(seq)` returns the concatenation of the string representations of these objects as a single string.

Examples.

- *Input:*

```
cat("abcd",3,"d")
```

Output:

```
"abcd3d"
```

- *Input:*

```
c:=5
cat("abcd",c,"e")
```

Output:

```
"abcd5e"
```

- *Input:*

```
purge(c)
cat(15,c,3)
```

Output:

```
"15c3"
```

6.3.12 Adding an object to a string: +

The '+' command can be used like `cat` (see Section 6.3.11 p.127), and the + operator is the infix version. (See Section 6.16.1 p.239 for other uses of + and '+'.)

- '+' takes one argument:
seq, a sequence of objects, at least one of which is a string.
- '+'(*seq*) returns the concatenation of the string representations of the objects in *seq*.

Warning.

+ is infix and '+' is prefixed.

Examples.

- *Input:*

```
'+'("abcd",3,"d")
```

or:

```
"abcd"+3+"d"
```

Output:

```
"abcd3d"
```

- *Input:*

```
c:=5
```

then:

```
"abcd"+c+"d"
```

or:

```
'+'("abcd",c,"d")
```

Output:

```
"abcd5d"
```


6.3.13 Transforming a real number into a string: cat +

The `cat` command (see Section 6.3.11 p.127) can also be used to transform a real number into a string, as can `+` (see Section 6.3.12 p.128).

If `cat` has a real number as an argument, the result will be a string.

Example.

Input:

```
cat(123)
```

Output:

```
"123"
```

Similarly, if you add a real number to an empty string, the result will be a string.

Example.

Input:

```
""+123
```

Output:

```
"123"
```

6.3.14 Transforming a string into a number: expr

The `expr` command transforms a string representing a valid `Xcas` statement into the actual statement.

- `expr` takes one argument:
`str`, a string corresponding to an `Xcas` statement.
- `expr(str)` evaluates the statement.

Examples.

- *Input:*

```
expr("a:=1")
```

Output:

```
1
```

Then:

Input:

```
a
```

Output:

```
1
```

In particular, `expr` can transform a string representing a number into the number (see Section 5.1 p.95).

- *Input:*

```
expr("123")
```

Output:

123

- *Input:*

```
expr("0123")
```

Output:

83

since 0123 represents a base 8 integer (see Section 6.4.1 p.130) and $1 \cdot 8^2 + 2 \cdot 8 + 3 = 83$.

- *Input:*

```
expr("0x12f")
```

Output:

303

since 0x12f represents a base 16 number and $1 \cdot 16^2 + 2 \cdot 16 + 15 = 303$.

- *Input:*

```
expr("123.4567")
```

Output:

123.4567

- *Input:*

```
expr("123e-5")
```

Output:

0.00123

6.4 Writing an integer in a different base

6.4.1 Writing an integer in base 2, 8 or 16

Integers are typically entered and displayed in base 10. You can also enter an integer in base 2 (binary), base 8 (octal) or base 16 (hexadecimal).

You can enter a number in base 2 by prefixing it with 0b; the remaining digits have to be 0 or 1 since it is binary.

Example.

Input:

0b101

Output:

5

since 101 in binary is the same as $1 \cdot 1 + 0 \cdot 2 + 1 \cdot 2^2 = 5$ in decimal.

You can enter a number in octal by prefixing it with 0 or 0o; the remaining digits have to be 0 through 7 since it is base 8.

Example.

Input:

0512

Output:

330

since 512 in base 8 is the same as $2 \cdot 1 + 1 \cdot 8 + 5 \cdot 8^2 = 330$ in decimal.

You can enter a number in hexademical by prefixing it with 0x; the remaining digits have to be 0 through 9 or a through f (where a is 10, b is 11, ..., f is 15).

Example.

Input:

0x2f3

Output:

755

since 2f3 in base 16 is the same as $3 \cdot 1 + 15 \cdot 16 + 2 \cdot 16^2 = 755$ in decimal.

You can have Xcas print integers in octal or hexadecimal, as well as the default decimal. To change the base used for display, you can click on the red CAS status button and choose from the **Integer basis** menu (see Section 3.5.7 p.73, item 14). If you have Xcas set to display in hexadecimal, you will get the following:

Input:

15

Output:

0xF

Input:

0x15

Output:

0x15

6.4.2 Writing an integer in an arbitrary base b : `convert`

The `convert` command does various kinds of conversions depending on the option given as the second argument (see Section 6.23.26 p.323). `convertir` is a synonym for `convert`.

One thing that `convert` can do is convert integers to arbitrary bases and back to the default base, both with the option `base`.

To convert an integer into the list of its “digits” in base b :

- `convert` takes three arguments:
 - n , an integer.
 - `base`, the symbol verbatim.
 - b , a positive integer, the value of the base.
- `convert(n ,base, b)` returns the list of digits of the integer n when written in base b . The list of digits will start with the $1s$ term, then the bs term, the b^2 term, etc.

Example.

Input:

```
convert(123,base,8)
```

Output:

```
[3, 7, 1]
```

To check the answer, input 0173 (see Section 6.4.1 p.130) or `horner(revlist([3,7,1]),8)` (see Section 6.27.19 p.363 and Section 6.40.15 p.474) or `convert([3,7,1],base,8)`. The result will be 123.

The base used for `convert` can be any integer greater than 1.

Example.

Input:

```
convert(142,base,12)
```

Output:

```
[10, 11]
```

To convert the its “digits” in base b into a base 10 integer:

- `convert` takes three arguments:
 - L , a list of integers representing the digits of the integer in base b , assumed to go in order of increasing significance.
 - `base`, the symbol verbatim.
 - b , a positive integer, the value of the base.
- `convert(L ,base, b)` returns the integer which, in base b , has the digits given in L .

Examples.

- *Input:*

```
convert([3,7,1],base,8)
```

Output:

123

- *Input:*

```
convert([10,11],base,12)
```

Output:

142

6.5 Integers (and Gaussian Integers)

Xcas can manage integers with unlimited precision, such as the following (see Section 6.6.1 p.158):

Input:

```
factorial(100)
```

Output:

```
9332621544394415268169923885626670049071596826438162
1468592963895217599993229915608941463976156518286253
6979208272237582511852109168640000000000000000000000000
```

Gaussian integers are numbers of the form $a + ib$, where a and b are in \mathbb{Z} . For most functions in this section, you can use Gaussian integers in place of integers.

6.5.1 GCD: gcd igcd Gcd

The `gcd` command finds the greatest common divisor (GCD) of a set of integers or polynomials. (See also Section 6.28.5 p.381 for polynomials.) It can be called with one or two arguments.

igcd is a synonym for *gcd*.

With one argument:

- `gcd` takes one argument:
seq, a sequence or list of integers or polynomials.
- `gcd(seq)` returns the GCD of the elements of *seq*.

Examples.

- *Input:*

```
gcd(18,15)
```

Output:

3

- *Input:*

`gcd(18,15,21,36)`

Output:

3

- *Input:*

`gcd([18,15,21,36])`

Output:

3

- *Input :*

`gcd(-5-12*i,11-10*i)`

Output:

$3 + 2i$

With two arguments:

- `gcd` takes two arguments:
 s and t , two lists of the same length containing integers or polynomials
 (alternatively, a matrix m with two rows whose elements are integers
 or polynomials).
- `gcd(s, t)` (or `gcd(m)`) returns the list whose k th element is the GCD
 of the k th elements of s and t (or the k th column of m).

Examples.

- *Input:*

`gcd([6,10,12],[21,5,8])`

or:

`gcd([[6,10,12],[21,5,8]])`

Output:

$[3, 5, 4]$

- Find the greatest common divisor of $4n + 1$ and $5n + 3$ when $n \in \mathbb{N}$.
Input:

`f(n):=gcd(4*n+1,5*n+3)`

then input:

```

essai(n):={
  local j,a,L;
  L:=NULL;
  for (j:=-n;j<n;j++) {
    a:=f(j);
    if (a!=1) {
      L:=L,[j,a];
    }
  }
  return L;
}

```

then input:

`essai(20)`

Output:

`[-16, 7], [-9, 7], [-2, 7], [5, 7], [12, 7], [19, 7]`

From this information, a reasonable conjecture would be that $\gcd(4n+1, 5n+3) = 7$ if $n = 7k - 2$ for some $k \in \mathbb{Z}$ and $\gcd(4n+1, 5n+3) = 1$ otherwise.

Since $\gcd(a, b) = \gcd(a, b - c \cdot a)$ for integers a, b and c ; we have $\gcd(4n+1, 5n+3) = \gcd(4n+1, 5n+3 - (4n+1)) = \gcd(4n+1, n+2) = \gcd(4n+1 - 4(n+2), n+2) = \gcd(-7, n+2) = \gcd(7, n+2)$, and so $\gcd(4n+1, 5n+3) = 7$ if 7 divides $n+2$, namely $n+2 = 7k$ or $n = 7k - 2$, and $\gcd(4n+1, 5n+3) = 1$ otherwise. This proves the conjecture.

The `Gcd` command is the inert form of `gcd`; namely, it evaluates to `gcd`, for later evaluation.

Examples.

- *Input:*

`Gcd(18, 15)`

Output:

`gcd(18, 15)`

- *Input:*

`eval(Gcd(18, 15))`

Output:

3

(See Section [6.12.1](#) p.203.)

6.5.2 GCD of a list of integers: `lgcd`

The `lgcd` command also finds the GCD of a list of integers or polynomials.

- `lgcd` takes one argument:
 L , a list of integers (or polynomials).
- `lgcd(L)` returns the GCD of all the integers (or polynomials) in the list L .

Example.

Input:

$$\text{lgcd}([18, 15, 21, 36])$$

Output:

$$3$$

Remark.

`lgcd` does not accept two lists as arguments (even if they have the same size).

6.5.3 The least common multiple: `lcm`

The `lcm` command finds the least common multiple (LCM) of a set of integers or polynomials. (See also Section 6.28.8 p.384 for polynomials.)

With one argument:

- `lcm` takes one argument:
 seq , a sequence or list of integers or polynomials.
- `lcm(seq)` returns the LCM of the elements of s .

Examples.

- *Input:*

$$\text{lcm}(18, 15)$$

Output:

$$90$$

- *Input:*

$$\text{lcm}(-5-12*i, 11-10*i)$$

Output:

$$-53 + 8i$$

- *Input:*

$$\text{lcm}(18, 15, 21, 36)$$

Output:

1260

- *Input:*

`lcm([18,15,21,36])`*Output:*

1260

With two arguments:

- `lcm` takes two arguments:
 s and t , two lists of the same length containing integers or polynomials (alternatively, a matrix m with two rows whose elements are integers or polynomials).
- `lcm(s, t)` (or `lcm(m)`) returns the list whose k th element is the LCM of the k th elements of s and t (or the k th column of m).

Example.*Input:*`lcm([6,10,12],[21,5,8])`*or:*`lcm([[6,10,12],[21,5,8]])`*Output:*

[42, 10, 24]

6.5.4 Decomposition into prime factors: ifactor

The `ifactor` command factors an integer into its prime factors. (Note that a prime factor of a Gaussian integer is only determined up to a factor of ± 1 or $\pm i$.)

- `ifactor` takes one argument:
 n , an integer or a list of integers.
- `ifactor(n)` returns n in factored form (or a list of the integers in factored form).

Examples.

- *Input:*

`ifactor(90)`*Output:* $5 \cdot 2 \cdot 3^2$

- *Input:*

```
ifactor(-90)
```

Output:

$$-5 \cdot 2 \cdot 3^2$$

- *Input:*

```
ifactor(14+23*i)
```

Output:

$$i(2-i)^2(5+2i)$$

- *Input:*

```
ifactor([36,52])
```

Output:

$$[2^2 \cdot 3^2, 13 \cdot 2^2]$$

6.5.5 List of prime factors: ifactors

The `ifactors` command decomposes an integer into prime factors.

- `ifactors` takes one argument:
 n , an integer or a list of integers.
- `ifactors(n)` decomposes the integer n (or the integers of the list) into prime factors, given as a list (or a list of lists) in which each prime factor is followed by its multiplicity.

Examples.

- *Input:*

```
ifactors(90)
```

Output:

$$[2, 1, 3, 2, 5, 1]$$

since $90 = 2^1 3^2 5^1$.

- *Input:*

```
ifactors(-90)
```

Output:

$$[-1, 1, 2, 1, 3, 2, 5, 1]$$

- *Input:*

```
ifactors(31+22*i)
```

Output:

```
[i, 1, 2 - i, 1, 4 - i, 2]
```

- *Input:*

```
ifactors([36,52])
```

Output:

$$\begin{bmatrix} 2 & 2 & 3 & 2 \\ 2 & 2 & 13 & 1 \end{bmatrix}$$

6.5.6 Matrix of factors: maple_ifactors

The `maple_ifactors` command decomposes an integer into prime factors, and returns the result in Maple syntax.

- `maple_ifactors` takes one argument:
 n , an integer or a list of integers.
- `maple_ifactors(n)` decomposes the integer n (or the integers of the list) into prime factors, given as a list following the Maple syntax; namely, a list starting with +1 or -1 (for the sign), then a matrix with 2 columns whose rows are the prime factors and their multiplicity (or a list of such lists).

Examples.

- *Input:*

```
maple_ifactors(90)
```

Output:

$$\left[1, \begin{bmatrix} 2 & 1 \\ 3 & 2 \\ 5 & 1 \end{bmatrix} \right]$$

- *Input:*

```
maple_ifactor([36,52])
```

Output:

$$\begin{bmatrix} 1 & \begin{bmatrix} 2 & 2 \\ 3 & 2 \end{bmatrix} \\ 1 & \begin{bmatrix} 2 & 2 \\ 13 & 1 \end{bmatrix} \end{bmatrix}$$

6.5.7 The divisors of a number: `idivis` `divisors`

The `idivis` command finds the divisors of an (ordinary) integer. `divisors` is a synonym for *idivis*.

- `idivis` takes one argument:
 n , an integer or list of integers.
- `idivis(n)` returns the list of the divisors of the integer n (or of a list of such lists).

Examples.

- *Input:*

```
idivis(36)
```

Output:

```
[1, 2, 3, 4, 6, 9, 12, 18, 36]
```

- *Input:*

```
idivis([36, 22])
```

Output:

```
[[1, 2, 3, 4, 6, 9, 12, 18, 36], [1, 2, 11, 22]]
```

6.5.8 The integer Euclidean quotient: `iquo` `intDiv` `div`

The quotient and remainder of ordinary integers a and b are respectively integers q and r , where $a = b * q + r$ and $0 \leq r < b$.

The quotient and remainder of Gaussian integers a and b are respectively Gaussian integers q and r where $r = a - b * q$ is as small as possible. It can be proven that r can be found so that $|r|^2 \leq |b|^2/2$.

The `iquo` command finds the integer quotient of two integers. `intDiv` is a synonym for *iquo*.

- `iquo` takes two arguments:
 a and b , integers.
- `iquo(a, b)` returns the quotient q of a and b .

Examples.

- *Input:*

```
iquo(148, 5)
```

Output:

```
29
```

- *Input:*

```
iquo(factorial(148),factorial(145)+2 )
```

Output:

3176375

- *Input:*

```
iquo(25+12*i,5+7*i)
```

Output:

$3 - 2i$

Here $r = a - b*q = -4 + i$ and $|-4 + i|^2 = 17 < |5 + 7*i|^2/2 = 74/2 = 37$

The *div* operator is the infix version of *iquo*.

Example.

Input:

```
148 div 5
```

Output:

29

6.5.9 The integer Euclidean remainder: `irem` `remain` `smod` `mods` `mod` `%`

The `irem` command finds the remainder of two integers (see Section 6.5.8 p.140).

`remain` is a synonym for `irem`.

- `irem` takes two arguments:
 a and b , integers.
- `irem(a, b)` returns the remainder r of a divided by b .

Examples.

- *Input:*

```
irem(148,5)
```

Output:

3

- *Input:*

```
irem(factorial(148),factorial(45)+2 )
```

Output:

111615339728229933018338917803008301992120942047239639312

- *Input:*

$$\text{irem}(25+12*i, 5+7*i)$$

Output:

$$-4 + i$$

Here $r = a - b*q = -4 + i$ and $|-4+i|^2 = 17 < |5+7*i|^2/2 = 74/2 = 37$

The **smod** command finds the symmetric remainder of two (ordinary) integers.

mods is a synonym for **smod**.

- **smod** takes two arguments:
 a and b , integers.
- **smod**(a, b) returns the symmetric remainder s of the Euclidean division of a and b ; namely, the value s with $a = b*q + s$ and $-b/2 < s \leq b/2$.

Example.

Input:

$$\text{smod}(148, 5)$$

Output:

$$-2$$

The **mod** operator is an infix operator which takes an integer to a modular integer.

% is a synonym for **mod**.

- **mod** has two operands: a and b , ordinary integers.
- $a \bmod b$ returns $r \% b$ in $\mathbb{Z}/b\mathbb{Z}$, where r is the remainder of the Euclidean division of the arguments a and b .

Example.

Input:

$$148 \bmod 5$$

or:

$$148 \% 5$$

Output:

$$(-2) \% 5$$

Note that the result $-2 \% 5$ is not an integer (-2) but an element of $\mathbb{Z}/5\mathbb{Z}$ (see Section 6.34 p.419 for the possible operations in $\mathbb{Z}/5\mathbb{Z}$).

6.5.10 Euclidean quotient and Euclidean remainder of two integers: `iquorem`

The `iquorem` command finds both the quotient and remainder of two integers (see Section 6.5.8 p.140).

- `iquorem` takes two arguments:
 a and b , integers.
- `iquorem(a,b)` returns the list $[q,r]$, where q is the quotient and r the remainder of a divided by b .

Examples.

- *Input:*

$$\text{iquorem}(148,5)$$

Output:

$$[29,3]$$

- *Input:*

$$\text{iquorem}(25+12*i,5+7*i)$$

Output:

$$[3 - 2i, -4 + i]$$

6.5.11 Test of evenness: `even`

The `even` command tests an integer to see if it is even. (A Gaussian integer $a + ib$ is even exactly when a and b are both even and odd otherwise.)

- `even` takes one argument:
 n , an integer.
- `even(n)` returns 1 if n is even and returns 0 if n is odd.

Examples.

- *Input:*

$$\text{even}(148)$$

Output:

$$1$$

- *Input:*

$$\text{even}(149)$$

Output:

$$0$$

- *Input:*

$$\text{even}(2+4*i)$$

Output:

$$1$$

6.5.12 Test of oddness: odd

The `odd` command tests an integer to see if it is odd.

- `odd` takes one argument:
 n , an integer.
- `odd(n)` returns 1 if n is odd and returns 0 if n is even.

Examples.

- *Input:*

```
odd(148)
```

Output:

```
0
```

- *Input:*

```
odd(149)
```

Output:

```
1
```

6.5.13 Test of pseudo-primality: is_pseudoprime

A *pseudo-prime* is a number with a large probability of being prime (cf. Rabin's Algorithm and Miller-Rabin's Algorithm in the Algorithmic part (menu **Help►Manuals►Programming**)). For numbers less than 10^{14} , pseudo-prime and prime are equivalent.

The `is_pseudoprime` command is a test for a pseudo-prime.

- `is_pseudoprime` takes one argument:
 n , an integer.
- `is_pseudoprime(n)` returns 0, 1 or 2.
 - If it returns 0, then n is not prime.
 - If it returns 1, then n is a prime.
 - If it returns 2, then n is pseudo-prime (most probably prime).

Examples.

- *Input:*

```
is_pseudoprime(100003)
```

Output:

```
1
```


- *Input:*

```
is_pseudoprime(9856989898997)
```

Output:

2

- *Input:*

```
is_pseudoprime(14)
```

Output:

0

- *Input:*

```
is_pseudoprime(9856989898997789789)
```

Output:

1

6.5.14 Test of primality: `is_prime` `isprime` `isPrime`

The `is_prime`, `isprime` and `isPrime` commands are tests for primality.

- `is_prime` takes one argument:
 n , an integer.
- `is_prime(n)` returns 1 if n is prime and 0 if n is not prime.

`isprime` and `isPrime` are the same as `is_prime`, except they return `true` or `false`.

Examples.

- *Input:*

```
is_prime(100003)
```

Output:

1

- *Input:*

```
isprime(100003)
```

Output:

true

- *Input:*

```
is_prime(98569898989987)
```

Output:

1

• *Input:*

`is_prime(14)`

Output:

0

• *Input:*

`isprime(14)`

Output:

false

You can use the command `pari("isprime",n,1)` (see Section 6.7.10 p.171) to get a primality certificate (see the documentation PARI/GP with the menu **Help►Manuals►PARI-GP**) and `pari("isprime",n,2)` to use the APRCL test.

Examples.

• *Input:*

`isprime(9856989898997789789)`

Output:

true

• *Input:*

`pari("isprime",9856989898997789789,1)`

Output:

$$\begin{pmatrix} 2 & 2 & 1 \\ 19 & 2 & 1 \\ 941 & 2 & 1 \\ 1873 & 2 & 1 \end{pmatrix}$$

which are the coefficients giving the proof of primality by the $p - 1$ Selfridge-Pocklington-Lehmer test.

6.5.15 The smallest pseudo-prime greater than n : nextprime

The `nextprime` command finds pseudo-primes larger than a given target.

- `nextprime` takes one argument:
 n , an integer.
- `nextprime(n)` returns the smallest pseudo-prime (or prime) greater than n .

Example.

Input:

```
nextprime(75)
```

Output:

79

6.5.16 The greatest pseudo-prime less than n : prevprime

The `prevprime` command finds pseudo-primes less than a given target.

- `prevprime` takes one argument:
 n , an integer greater than 2.
- `prevprime(n)` returns the largest pseudo-prime (or prime) less than n .

Example.

Input:

```
prevprime(75)
```

Output:

73

6.5.17 The n th pseudo-prime number: ithprime

The `ithprime` command finds pseudo-primes.

- `ithprime` takes one argument:
 n , a positive integer.
- `ithprime(n)` returns the n th pseudo-prime number.

Examples.

- *Input:*

```
ithprime(75)
```

Output:

379

- *Input:*

```
ithprime(k) $(k=1..20)
```

Output:

```
2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71
```

6.5.18 The number of pseudo-primes less than or equal to n : `nprimes`

The `nprimes` command counts the number of pseudo-primes.

- `nprimes` takes one argument:
 n , a non-negative integer.
- `nprimes(n)` returns the number of pseudo-primes (or primes) less than or equal to n .

Examples.

- *Input:*

```
nprimes(5)
```

Output:

```
3
```

- *Input:*

```
nprimes(10)
```

Output:

```
4
```

6.5.19 Bézout's Identity: `iegcd` `igcdex`

Bézout's Identity states that for any integers a and b , there exist integers u and v such that $\gcd(a, b) = au + bv$. The `iegcd` command computes the coefficients u and v .

`igcdex` is a synonym for `iegcd`.

- `iegcd` takes two arguments:
 a and b , integers.
- `iegcd(a, b)` returns the list $[u, v, d]$, where $au + bv = d$ and $d = \gcd(a, b)$.

Example.

Input:

```
iegcd(48, 30)
```

Output:

```
[2, -3, 6]
```

In other words:

$$2 \cdot 48 + (-3) \cdot 30 = 6$$

6.5.20 Solving $au + bv = c$ in \mathbb{Z} : `iabcuv`

The `iabcuv` solves a linear Diophantine equation in two variables.

- The `iabcuv` command takes three arguments:
 a , b and c , integers.
- `iabcuv(a,b,c)` returns the list $[u,v]$ where $au + bv = c$.

Note that c must be a multiple of $\gcd(a,b)$ for the existence of a solution.

Example.

Input:

```
iabcuv(48,30,18)
```

Output:

```
[6,-9]
```

6.5.21 Chinese remainders: `ichinrem` `ichrem` `chrem`

The Chinese Remainder Theorem states that if p_1, p_2, \dots, p_n are relatively prime, then for any integers a_1, a_2, \dots, a_n there is a number c such that $c = a_1 \pmod{p_1}$, $c = a_2 \pmod{p_2}, \dots, c = a_n \pmod{p_n}$. The `ichinrem` command will find this value of c .

`ichrem` is a synonym for `ichinrem`.

- `ichinrem` takes one or more arguments:
Each argument is a pair of integers a_k and p_k either as a list $[a_k, p_k]$ or as a modular integer $a_k \% p_k$.
- `ichinrem([a1, p1], [a2, p2], ..., [an, pn])` if possible returns a list $[c, L]$, where $L = \text{lcm}(p_1, p_2, \dots, p_n)$ and c satisfies $c = a_k \pmod{p_k}$ for $k = 1, \dots, n$.

Note that any multiple of $L = \text{lcm}(p_1, p_2, \dots, p_n)$ can be added to c and the equalities will still be true. If the p_k are relatively prime, then by the Chinese remainder theorem a solution c will exist; what's more, any two solutions will be congruent modulo the product of the p_k s.

If all of the arguments are given as modular integers, then the result will also be given as a modular integer $c \% l$.

The `chrem` command does the same thing as `ichinrem`, but the input is given in a different form.

- `chrem` takes two arguments:
 $[a_1, \dots, a_n]$ and $[p_1, \dots, p_n]$, lists of integers of the same size.
- `chrem([a1, ..., an], [p1, ..., pn])` returns $[c, L]$, as for `ichinrem`.

BE CAREFUL with the order of the parameters, indeed:

```
chrem([a,b],[p,q])=ichrem([a,p],[b,q])=ichinrem([a,p],[b,q])
```

Examples.

- Solve:

$$\begin{aligned}x &= 3 \pmod{5} \\ x &= 9 \pmod{13}\end{aligned}$$

Input:

```
ichinrem([3,5],[9,13])
```

or:

```
ichrem([3,5],[9,13])
```

Output:

```
[48,65]
```

so $x=48 \pmod{65}$

You can also input:

```
ichrem(3%5,9%13)
```

Output:

```
(-17)%65
```

(note that $48 = -17 \pmod{65}$).

Recalling that `chrem` takes its arguments in a different form, you can also enter:

Input:

```
chrem([3,9],[5,13])
```

Output:

```
[48,65]
```

- Solve:

$$\begin{aligned}x &= 3 \pmod{5} \\ x &= 4 \pmod{7} \\ x &= 1 \pmod{9}\end{aligned}$$

Input:

```
ichinrem([3,5],[4,7],[1,9])
```

Output:

```
[298,315]
```

hence $x=298 \pmod{315}$

Alternative input:

```
ichinrem([3%5,4%7,1%9])
```

Output:

$$(-17) \% 315$$

(note that $298 = -17 \pmod{315}$).

Again, with the arguments in a different form, you can also enter:

Input:

$$\text{chrem}([3,4,1],[5,7,9])$$

Output:

$$[298,315]$$

Remark.

These three commands, `ichinrem`, `ichrem` and `chrem`, may also be used to find the coefficients of a polynomial whose equivalence classes are known modulo several integers by using polynomials with integer coefficients instead of integers for the a_k .

For example, to find $ax + b$ modulo $315 = 5 \times 7 \times 9$ under the assumptions

$$a = 3 \pmod{5}$$

$$a = 4 \pmod{7}$$

$$a = 1 \pmod{9}$$

and

$$b = 1 \pmod{5}$$

$$b = 2 \pmod{7}$$

$$b = 3 \pmod{9}$$

Example.

Input:

$$\text{ichinrem}((3x+1)\%5,(4x+2)\%7,(x+3)\%9)$$

Output:

$$((-17) \% 315)x + 156 \% 315$$

hence $a = -17 \pmod{315}$ and $b = 156 \pmod{315}$.

As before, `chrem` takes the same input in a different format.

Input:

$$\text{chrem}([3x+1,4x+2,x+3],[5,7,9])$$

Output:

$$[298x + 156, 315]$$

(note that $298 = -17 \pmod{315}$).

6.5.22 Solving $a^2 + b^2 = p$ in \mathbb{Z} : `pa2b2`

Any prime number congruent to 1 modulo 4 can be written as a sum of two squares. The `pa2b2` command finds such a decomposition.

- `pa2b2` takes one argument:
 p , a prime number which is congruent to 1 modulo 4.
- `pa2b2(p)` returns a list of integers $[a, b]$, where $p = a^2 + b^2$.

Example.

Input:

`pa2b2(17)`

Output:

`[4, 1]`

indeed, $17 = 4^2 + 1^2$.

6.5.23 Solving Diophantine equations: `isolve`

The `isolve` command attempts to solve the given equations over the integers. Note that it automatically solves for all of the indeterminates present in the equations.

- `isolve` takes one mandatory argument: an equation or list of equations.
- `isolve` also takes the following optional arguments:
 - a (sequence or list of) symbol(s), which are used as the names for global variables present in the solution, can be passed as the second argument. These names default to `_Z0, _Z1, ...` for general integers and to `_N0, _N1, ...` for positive integers.
 - `seq=false`, which makes `isolve` return only particular/fundamental solution(s) found by the solver. By default, `seq=true`, which makes `isolve` return sequences (classes) of solutions whenever possible.
- `isolve` can solve the following types of equations:
 - (systems of) linear equation(s)
 - general quadratic equations with two indeterminates
 - equations of the type $Q(x, y, z) = 0$, where Q is a ternary quadratic form
 - equations of the type $f(x) = g(y)$, where $f, g \in \mathbb{Z}[X]$ are monic polynomials with degrees m and n such that $\gcd(m, n) > 1$ and $f(x) - g(y)$ is irreducible in $\mathbb{Q}[X, Y]$

Examples.

- Linear equations and systems can be solved.

Input:

```
isolve(5x+42y+8=0)
```

Output:

$$[x = -10 + 42_Z0, y = 1 - 5_Z0]$$

Input:

```
isolve([x+y-z=4,x-2y+3z=3],m)
```

Output:

$$[x = m, y = -4m + 15, z = -3m + 11]$$

- Here we find the general solution to Pell-type equation $x^2 - 23y^2 = 1$.

Input:

```
sol:=isolve(x^2-23y^2=1,n)
```

Output:

$$\left[x = \frac{(24 + 5\sqrt{23})^n + (24 - 5\sqrt{23})^n}{2}, y = \frac{(24 + 5\sqrt{23})^n - (24 - 5\sqrt{23})^n}{2\sqrt{23}} \right]$$

To check that it is indeed the solution, enter:

```
simplify(subs(x^2-23y^2-1,sol))
```

Output:

0

Now to obtain e.g. the first four solutions, enter:

```
simplify(apply(unapply(apply(rhs,sol),n),[1,2,3,4]))
```

Output:

$$[[24, 5], [1151, 240], [55224, 11515], [2649601, 552480]]$$

To obtain only the fundamental solution, enter:

```
isolve(x^2-23y^2=1,seq=false)
```

Output:

$$[x = 24, y = 5]$$

- The following two examples demonstrate solving quadratic equations with two indeterminates.

Input:

`isolve(x^2-5x*y+4y^2=16)`

Output:

$$\begin{bmatrix} x = -4, & y = -5 \\ x = 0, & y = -2 \\ x = 4, & y = 0 \\ x = 10, & y = 2 \\ x = 21, & y = 5 \\ x = 4, & y = 5 \\ x = 0, & y = 2 \\ x = -4, & y = 0 \\ x = -10, & y = -2 \\ x = -21, & y = -5 \end{bmatrix}$$

Input:

`isolve(x^2-3x*y+y^2-x=2,n)`

Output:

$$\begin{aligned} [x = & \frac{\left(\frac{\sqrt{5}-3}{2}\right)^n (-15\sqrt{5}-33)}{10} + \frac{\left(\frac{-\sqrt{5}-3}{2}\right)^n (15\sqrt{5}-33)}{10} - \frac{2}{5}, \\ y = & \frac{\left(\frac{\sqrt{5}-3}{2}\right)^n (-3\sqrt{5}-6)}{5} + \frac{\left(\frac{-\sqrt{5}-3}{2}\right)^n (3\sqrt{5}-6)}{5} - \frac{3}{5}] \end{aligned}$$

Input:

`isolve(8x^2-24x*y+18y^2+5x+7y+16=0)`

Output:

$$\begin{bmatrix} x = -174_Z1^2 + 17_Z1 - 2, & y = -116_Z1^2 + 21_Z1 - 2 \\ x = -174_Z1^2 + 41_Z1 - 4, & y = -116_Z1^2 + 37_Z1 - 4 \end{bmatrix}$$

- Integral zeros of ternary quadratic forms can be found.

Input:

`isolve(x^2+11y^2+6x*y-3z^2=0,a,b,c)`

Output:

$$[x = c(-44a^2 + 12b^2), y = c(13a^2 - 3b^2 - 6ab), z = c(-11a^2 - 3b^2 + 2ab)]$$

The components of the above solution can be divided by the GCD of $-44a^2 + 12b^2$, $13a^2 - 3b^2 - 6ab$, and $-11a^2 - 3b^2 + 2ab$, thus producing a parametrization for the pairwise-coprime solutions given $c = 1$.

- Certain polynomial equations of the type $f(x) = g(y)$ can be fully solved, as in the following example.

Input:

```
isolve(x^2-3x+5=y^8-y^7+9y^6-7y^5+4y^4-y^3)
```

Output:

$$\left[\begin{array}{ll} x = -3, & y = -1 \\ x = 6, & y = -1 \\ x = 0, & y = 1 \\ x = 3, & y = 1 \\ x = 660, & y = 5 \\ x = -657, & y = 5 \end{array} \right]$$

The above list contains all integer solutions to the given equation.

6.5.24 The Euler indicatrix: euler phi

The Euler phi function (also called the Euler totient function) finds the number of positive integers less than a given integer and relatively prime. The `euler` command computes the Euler phi function.

- `euler` takes one argument:
 n , a non-negative integer.
- `euler(n)` returns the number of integers larger than 1, less than n and relatively prime to n .

Example.

Input:

```
euler(21)
```

Output:

```
12
```

In other words the set of integers less than 21 and coprime with 21, $\{2, 4, 5, 7, 8, 10, 11, 13, 15, 16, 17, 19\}$, has 12 elements.

The little Fermat theorem states:

If p is a prime number, then for any integer a , $a^{p-1} = 1 \pmod{p}$.

Euler introduced his phi function to generalize the little Fermat theorem:

If a and n are relatively prime, then $a^{\text{euler}(n)} = 1 \pmod{n}$.

Example.

Input:

```
powmod(5,12,21)
```

(see section Section 6.34.10 p.424)

Output:

6.5.25 Legendre symbol: `legendre_symbol`

If n is prime, the Legendre symbol of a is written $\left(\frac{a}{n}\right)$ and defined by:

$$\left(\frac{a}{n}\right) = \begin{cases} 0 & \text{if } a \equiv 0 \pmod{n} \\ 1 & \text{if } a \not\equiv 0 \pmod{n} \text{ and if } a \equiv b^2 \pmod{n} \\ -1 & \text{if } a \not\equiv 0 \pmod{n} \text{ and if } a \not\equiv b^2 \pmod{n} \end{cases}$$

The Legendre symbol satisfies the following properties.

- If n is prime:

$$a^{\frac{n-1}{2}} \equiv \left(\frac{a}{n}\right) \pmod{n}$$

-

$$\begin{aligned} \left(\frac{p}{q}\right) \cdot \left(\frac{q}{p}\right) &= (-1)^{\frac{p-1}{2}} \cdot (-1)^{\frac{q-1}{2}} \text{ if } p \text{ and } q \text{ are odd and positive} \\ \left(\frac{2}{p}\right) &= (-1)^{\frac{p^2-1}{8}} \\ \left(\frac{-1}{p}\right) &= (-1)^{\frac{p-1}{2}} \end{aligned}$$

The `legendre_symbol` command computes the Legendre symbol.

- `legendre_symbol` takes two arguments:
 a and n , integers.
- `legendre_symbol(a,n)` returns the Legendre symbol $\left(\frac{a}{n}\right)$.

Examples.

- *Input:*

```
legendre_symbol(26,17)
```

Output:

1

- *Input:*

```
legendre_symbol(27,17)
```

Output:

-1

- *Input:*

```
legendre_symbol(34,17)
```

Output:

0

6.5.26 Jacobi symbol: `jacobi_symbol`

The Jacobi symbol is a generalization of the Legendre symbol $\left(\frac{a}{n}\right)$ for when n isn't prime. Let

$$n = p_1^{\alpha_1} \cdots p_k^{\alpha_k}$$

be the prime factorization of n . The Jacobi symbol of a is defined by:

$$\left(\frac{a}{n}\right) = \left(\frac{a}{p_1}\right)^{\alpha_1} \cdots \left(\frac{a}{p_k}\right)^{\alpha_k}$$

Where the left hand side is the Jacobi symbol and the right hand side contains Legendre symbols. The `jacobi_symbol` command computes the Jacobi symbol.

- `jacobi_symbol` takes two arguments:
 a and n , integers.
- `jacobi_symbol(a,n)` returns the Jacobi symbol $\left(\frac{a}{n}\right)$.

Examples.

- *Input:*

```
jacobi_symbol(25,12)
```

Output:

1

- *Input:*

```
jacobi_symbol(35,12)
```

Output:

-1

- *Input:*

```
jacobi_symbol(33,12)
```

Output:

0

6.5.27 Listing all compositions of an integer into k parts: `icomp`

A composition of a positive integer n is an ordered set of non-negative integers which sum to n . For example, three compositions of 4 are

$$4 = 1 + 3$$

$$4 = 3 + 1$$

$$4 = 1 + 1 + 2$$

These compositions have two, two and three elements, respectively. The `icomp` command finds all compositions of an integer with a given number of elements.

- `icomp` accepts two mandatory arguments and one optional argument:
 - n , a positive integer.
 - k , a positive integer not larger than n .
 - Optionally, either `zeros=true` or `zeros=false`.
- `icomp(n, k , <,zeros=bool>)` returns the list of all compositions of n into k parts, where a part can be 0. This is equivalent to the optional argument with `bool` equal to `true`. With `bool` equal to `false`, `icomp(n, k , zeros=false)` returns the list of all compositions of n into k parts, where each part is nonzero (positive).

Examples.

- *Input:*

`icomp(4,2)`

Output:

$$\begin{bmatrix} 4 & 0 \\ 3 & 1 \\ 2 & 2 \\ 1 & 3 \\ 0 & 4 \end{bmatrix}$$

- *Input:*

`icomp(6,3,zeros=false)`

Output:

$$\begin{bmatrix} 4 & 1 & 1 \\ 3 & 2 & 1 \\ 2 & 3 & 1 \\ 1 & 4 & 1 \\ 3 & 1 & 2 \\ 2 & 2 & 2 \\ 1 & 3 & 2 \\ 2 & 1 & 3 \\ 1 & 2 & 3 \\ 1 & 1 & 4 \end{bmatrix}$$

6.6 Combinatorial analysis

6.6.1 Factorial: `factorial !`

The `factorial` command computes the factorial of a number. The postfix operator `!` is equivalent.

- `factorial` takes one argument:
 - n , an integer.

- `factorial(n)` returns $n!$.

Example.

Input:

```
factorial(10)
```

or:

```
10!
```

Output:

```
3628800
```

The Γ function (see Section 6.8.13 p.183) can be used to extend the factorial function to complex numbers. The Γ function is defined for all complex numbers except for zero and the negative integers, and it satisfies $\Gamma(n+1) = n!$ for all non-negative integers n . So the factorial can be extended to all complex numbers except the negative integers by $n! = \Gamma(n+1)$.

Examples.

- *Input:*

```
factorial(1/2)
```

Output:

$$\frac{\sqrt{\pi}}{2}$$

- *Input:*

```
factorial(i)
```

Output:

$$0.5 - 0.2i$$

6.6.2 Binomial coefficients: `binomial` `comb` `nCr`

The `comb` command computes the binomial coefficients. `nCr` is a synonym for `comb`.

- `comb` takes two arguments:
 n and p , integers.
- `comb(n,p)` returns $\binom{n}{p} = C_n^p$.

Example.

Input:

```
comb(5,2)
```

Output:

10

Remark.

The `binomial` command (see Section 9.4.3 p.761) can also compute the binomial coefficients, but unlike `comb` and `nCr` it can take an optional third argument, a real number a , to compute the binomial distribution. In this case `binomial(n, p, a)` returns $\binom{n}{p} a^p (1 - a)^{n-p}$, the probability of p successes in n independent Bernoulli trials, where each trial has a probability a of success.

Example.

Input:

`binomial(5,2,0.5)`

Output:

0.3125

6.6.3 Permutations: `perm nPr`

The `perm` command computes numbers of permutations. `nPr` is a synonym for `perm`.

- `perm` takes two arguments:
 n and p , integers.
- `perm(n, p)` returns P_n^p , the number of permutations of n objects taken p at a time.

Example.

Input:

`perm(5,2)`

Output:

20

6.6.4 Wilf-Zeilberger pairs: `wz_certificate`

The Wilf-Zeilberger certificate $R(n, k)$ is used to prove the identity

$$\sum_k U(n, k) = \text{Cres}(n)$$

for some constant C (typically 1) whose value can be determined by evaluating both sides for some value of k . To see how that works, note that the above identity is equivalent to

$$\sum_k F(n, k)$$

being constant, where $F(n, k) = U(n, k)/\text{res}(n)$. The Wilf-Zeilberger certificate is a rational function $R(n, k)$ that make $F(n, k)$ and $G(n, k) = R(n, k)F(n, k)$ a Wilf-Zeilberger pair, meaning

- $F(n+1, k) - F(n, k) = G(n, k+1) - G(n, k)$ for integers $n \geq 0, k$.
- $\lim_{k \rightarrow \pm\infty} G(n, k) = 0$ for each $n \geq 0$.

To see how this helps, adding the first equation from $k = -M$ to $k = N$ gives you $\sum_{k=-M}^N (F(n+1, k) - F(n, k)) = \sum_{k=-M}^N (G(n, k+1) - G(n, k))$. The right-hand side is a telescoping series, and so the equality can be written

$$\sum_{k=-M}^N F(n+1, k) - \sum_{k=-M}^N F(n, k) = G(n, N+1) - G(n, -M).$$

Taking the limit as $N, M \rightarrow \infty$ and using the second condition of Wilf-Zeilberger pairs, you get

$$\sum_k F(n+1, k) = \sum_k F(n, k)$$

and so $\sum_k F(n, k)$ does not depend on n , and so is a constant.

The `wz_certificate` command computes Wilf-Zeilberger pairs.

- `wz_certificate` takes four arguments:
 - $U(n, k)$, an expression in two variables.
 - $res(k)$, an expression in one of the variables.
 - n and k , the variables.
- `wz_certificate($U(n, k), res(k), n, k$)` returns the Wilf-Zeilberger certificate $R(n, k)$ for the identity $\sum_{k=-\infty}^{infy} U(n, k) = res(n)$.

Example.

To show

$$\sum_k (-1)^k \binom{n}{k} \binom{2k}{k} 4^{n-k} = \binom{2n}{n} :$$

Input:

```
wz_certificate((-1)^k*comb(n,k)*comb(2k,k)*4^(n-k), comb(2n,n), n, k)
```

Output:

$$\frac{2k-1}{2n+1}$$

This means that $R(n, k) = (2k-1)/(2n+1)$ is a Wilf-Zeilberger certificate; in other words $F(n, k) = (-1)^k \binom{n}{k} \binom{2k}{k} 4^{n-k} / \binom{2n}{n}$ and $G(n, k) = R(n, k)F(n, k)$ are a Wilf-Zeilberger pair. So $\sum_k F(n, k)$ is a constant. Since $F(0, 0) = 1$ and $F(0, k) = 0$ for $k > 0$, $\sum_k F(0, k) = 1$ and so $\sum_k F(n, k) = 1$ for all n , showing

$$\sum_k (-1)^k \binom{n}{k} \binom{2k}{k} 4^{n-k} = \binom{2n}{n}.$$

6.7 Rational numbers

6.7.1 Transform a floating point number into a rational: `exact` `float2rational`

Rational numbers can be approximated by floating point numbers, but since floating point numbers are not exact, they can't typically be converted back to the original rational number. However, the `float2rational` command will try convert a floating point to a nearby rational number.

`exact` is a synonym for `float2rational`.

- `float2rational` takes one argument:
 d , a floating point number.
- `float2rational(d)` returns a rational number q close to d ; namely such that $|d - q| < \text{epsilon}$, where `epsilon` is defined in the `cas` configuration (`Cfg` menu, see Section 3.5.7 p.73, item 9) or with the `cas_setup` command (see Section 3.5.10 p.78).

Examples.

- *Input:*

```
float2rational(0.3670520231)
```

Output (when `epsilon=1e-10`):

$$\frac{127}{346}$$

- *Input:*

```
evalf(363/28)
```

Output:

```
12.9642857143
```

- *Input:*

```
float2rational(12.9642857143)
```

Output:

$$\frac{363}{28}$$

- If two representations are mixed, for example:

Input:

```
1/2+0.7
```

the rational is converted to a float.

Output:

```
1.2
```

- *Input:*

`1/2+float2rational(0.7)`

Output:

$$\frac{6}{5}$$

6.7.2 Integer and fractional part: `propfrac` `propFrac`

Rational numbers are often broken up into integer and fractional parts, where the fractional part has absolute value less than 1; i.e., the absolute value of the top integer is smaller than that of the bottom integer. Such a fraction is called a *proper* fraction. The `propfrac` command writes a fraction as an integer plus a proper fraction.

`propFrac` is a synonym for `propfrac`.

- `propfrac` takes one argument:
 r , a rational number.
- `propfrac(r)` returns

$$q + \frac{r}{b} \quad \text{with } 0 \leq r < b$$

where $r = \frac{a}{b}$ is in lowest terms and $a = bq + r$.

(For rational expressions, see Section 6.32.8 p.416.)

Examples.

- *Input:*

`propfrac(42/15)`

Output:

$$2 + \frac{4}{5}$$

- *Input:*

`propfrac(43/12)`

Output:

$$3 + \frac{7}{12}$$

6.7.3 Numerator of a fraction after simplification: `numer` `getNum`

The `numer` command finds the numerator of a fraction.
`getNum` is a synonym for `numer`.

- `numer` takes one argument:
 r , a fraction.
- `numer(r)` returns the numerator of r after it has been reduced to lowest terms. (For rational expressions, see Section 6.32.2 p.413 and Section 6.32.1 p.413.)

Examples.

- *Input:*

```
numer(42/12)
```

or:

```
getNum(42/12)
```

Output:

7

- To avoid simplification, the argument must be quoted (see Section 6.12.4 p.205).
(For rational fractions see 6.32.1).

Input:

```
numer('42/12')
```

or:

```
getNum('42/12')
```

Output:

42

6.7.4 Denominator of a fraction after simplification: `denom` `getDenom`

The `denom` command finds the denominator of a fraction.
`getDenom` is a synonym for `denom`.

- `denom` takes one argument:
 r a fraction.
- `denom(r)` returns the denominator of r after it has been reduced to lowest terms. (For rational expressions see Section 6.32.4 p.414).

Example.*Input:*

```
denom(42/12)
```

or:

```
getDenom(42/12)
```

Output:

```
2
```

To avoid simplification, the argument must be quoted (see Section 6.12.4 p.205).

(For rational expressions see Section 6.32.3 p.414).

Input:

```
denom('42/12')
```

or:

```
getDenom('42/12')
```

Output:

```
12
```

6.7.5 Numerator and denominator of a fraction: f2nd fxnd

The `f2nd` command finds the numerator and denominator of a fraction.

`fxnd` is a synonym for `f2nd`.

- `f2nd` takes one argument:
 r , a fraction.
- `f2nd(r)` returns the list of the numerator and denominator of r after it has been reduced to lowest terms. (For rational expressions see Section 6.32.5 p.415).

Example.*Input:*

```
f2nd(42/12)
```

Output:

```
[7, 2]
```

6.7.6 Simplifying a pair of integers: simp2

The `simp2` command reduces a fraction to lowest terms, where the fraction is given as a separate numerator and denominator. (See also Section 6.32.6 p.415.)

- `simp2` takes one or two arguments:
 $[a, b]$, a list of two integers or simply the two integers a, b .

- `simp2([a,b])` or `simp2(a,b)` returns the integers after they have been divided by their greatest common divisor; i.e., the corresponding fraction will be in lowest terms.

Examples.

- *Input:*

`simp2(18,15)`

Output:

[6, 5]

- *Input:*

`simp2([42,12])`

Output:

[7, 2]

6.7.7 Continued fraction representation of a real: `dfc`

Any real number a can be written as a continued fraction:

$$a = a_0 + \frac{1}{a_1 + \frac{1}{a_2 + \frac{1}{a_3 + \dots}}}$$

, which is often abbreviated $[a_0; a_1, a_2, a_3, \dots]$. The `dfc` command writes a real number as a continued fraction.

- `dfc` takes one mandatory argument and one optional argument:
 - a , a real number.
 - Optionally, n an integer or *epsilon*, a positive real number.
- `dfc(a)` returns the list of the continued fraction representation of a with precision `epsilon`, which is given by Section 3.5.7 p.73, item 9.
- `dfc(a,epsilon)` returns the list of the continued fraction representation which approximates a or `evalf(a)` with the specified precision `epsilon`.
- `dfc(a,n)` returns the list of the continued fraction representation of a of order n .

Remarks.

- The `convert` command with the option `confrac` (see Section 6.23.26 p.323) has a similar functionality: in that case the value of `epsilon` is the value defined in the `cas` configuration and the answer may be stored in an optional third argument.

- If the last element of the result is a list, the representation is ultimately periodic, and the last element is the period. It means that the real is a root of an equation of order 2 with integer coefficients. So If $\text{dfc}(a)=[a_0, a_1, a_2, [b_0, b_1]]$ then:

$$a = a_0 + \frac{1}{a_1 + \frac{1}{a_2 + \frac{1}{b_0 + \frac{1}{b_1 + \frac{1}{b_0 + \dots}}}}}$$

- if the last element of the result is not an integer, it represents a remainder r ($a = a_0 + 1/\dots + 1/an + 1/r$). So if $\text{dfc}(a)=[a_0, a_1, a_2, r]$ then:

$$a = a_0 + \frac{1}{a_1 + \frac{1}{a_2 + \frac{1}{r}}}$$

Be aware that this remainder has lost most of its accuracy.

Examples.

- *Input:*

`dfc(sqrt(2), 5)`

Output:

`[1, 2, [2]]`

- *Input:*

`dfc(evalf(sqrt(2)), 1e-9)`

or:

`dfc(sqrt(2), 1e-9)`

Output:

`[1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]`

- *Input:*

`convert(sqrt(2), confrac, 'dev')`

Output (if in the cas configuration epsilon=1e-9):

`[1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]`

and `[1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]` is stored in `dev`.

- *Input:*

`dfc(9976/6961, 5)`

Output:

$$\left[1, 2, 3, 4, 5, \frac{43}{7}\right]$$

Input (to verify):

$$1+1/(2+1/(3+1/(4+1/(5+7/43))))$$

Output:

$$\frac{9976}{6961}$$

- *Input:*

$$\text{convert}(9976/6961, \text{confrac}, '1')$$

Output (if in the cas configuration epsilon=1e-9):

$$[1, 2, 3, 4, 5, 6, 7]$$

and $[1, 2, 3, 4, 5, 6, 7]$ is stored in 1.

- *Input:*

$$\text{dfc}(\text{pi}, 5)$$

Output:

$$\left[3, 7, 15, 1, 292, \frac{-113\pi + 355}{33102\pi - 103993}\right]$$

- *Input:*

$$\text{dfc}(\text{evalf}(\text{pi}), 5)$$

Output (if floats are hardware floats, e.g. for Digits=12):

$$[3, 7, 15, 1, 292, 1.57581843574]$$

- *Input:*

$$\text{dfc}(\text{evalf}(\text{pi}), 1\text{e-}9)$$

or:

$$\text{dfc}(\text{pi}, 1\text{e-}9)$$

or (if in the cas configuration epsilon=1e-9):

$$\text{convert}(\text{pi}, \text{confrac}, '11')$$

Output:

$$[3, 7, 15, 1, 292]$$

and $[3, 7, 15, 1, 292]$ is stored in 11.

6.7.8 Transforming a continued fraction representation into a real: `dfc2f`

The `dfc2f` command transforms a continued fraction into a real number.

- `dfc2f` takes one argument:
 L , a list representing a continued fraction, which can be:
 - a list of integers for a rational number.
 - a list whose last element is a list for an ultimately periodic representation, i.e. a quadratic number, that is a root of a second order equation with integer coefficients.
 - a list with a remainder r as last element ($a = a_0 + 1/\dots + 1/a_n + 1/r$).
- `dfc2f(L)` returns the rational number or the quadratic number whose continued fraction representation is L .

Examples.

- *Input:*

`dfc2f([1,2,[2]])`

Output:

$$\frac{1}{\frac{1}{\sqrt{2}+1} + 2} + 1$$

After simplification with `normal`:

$$\sqrt{2}$$

- *Input:*

`dfc2f([1,2,3])`

Output:

$$\frac{10}{7}$$

- *Input:*

`normal(dfc2f([3,3,6,[3,6]]))`

Output:

$$\sqrt{11}$$

- *Input:*

`dfc2f([1,2,3,4,5,6,7])`

Output:

$$\frac{9976}{6961}$$

Input (to verify):

$$1+1/(2+1/(3+1/(4+1/(5+1/(6+1/7))))))$$

Output:

$$\frac{9976}{6961}$$

• *Input:*

$$\text{dfc2f}([1, 2, 3, 4, 5, 43/7])$$

Output:

$$\frac{9976}{6961}$$

Input (to verify):

$$1+1/(2+1/(3+1/(4+1/(5+7/43))))$$

Output:

$$\frac{9976}{6961}$$

6.7.9 The n -th Bernoulli number: `bernoulli`

The Bernoulli polynomial B_n is defined by:

$$B_0 = 1, \quad B_n'(x) = nB_{n-1}(x), \quad \int_0^1 B_n(x)dx = 0$$

The n th Bernoulli number is $B_n = B_n(0)$, and is also given by the formula:

$$\frac{t}{e^t - 1} = \sum_{n=0}^{+\infty} \frac{B(n)}{n!} t^n$$

The `bernoulli` command computes the Bernoulli numbers.

- `bernoulli` takes one argument:
 n , an integer.
- `bernoulli(n)` returns the n -th Bernoulli number, B_n .

Example.

Input:

$$\text{bernoulli}(6)$$

Output:

$$\frac{1}{42}$$

6.7.10 Accessing to PARI/GP commands: `pari`

PARI/GP (<https://pari.math.u-bordeaux.fr/>) is a computer algebra system which focuses on number theory. `Xcas` can use the PARI/GP functions with the `pari` command.

The arguments of `pari` depends on the PARI/GP function it is using.

- `pari` with a string as first argument (the PARI command name) executes the corresponding PARI command with the remaining arguments. For example `pari("weber",1+i)` executes the PARI command `weber(1+i)`.
- `pari` without any argument exports all PARI/GP functions to `Xcas` with the prefix `pari_`. If the name of a PARI function is not also the name of an `Xcas` command, that function will also be exported without the prefix.

For example, after calling `pari()`, the commands `pari_weber(1+i)` and `weber(1+i)` will execute the PARI command `weber(1+i)`.

The documentation of PARI/GP is available with the menu **Help►Manuals**.

6.8 Real numbers

6.8.1 Evaluating a real at a given precision: `evalf` Digits

A real number is an exact number and its numeric evaluation at a given precision is a floating number represented in base 2. The precision of a floating number is the number of bits of its mantissa, which is at least 53 (hardware float numbers, also known as **double**).

Floating numbers are displayed in base 10 with a number of digits controlled by the user either by assigning the **Digits** variable or by modifying the Cas configuration (see Section 3.5.7 p.73, item 8). By default **Digits** is equal to 12.

The number of digits displayed controls the number of bits of the mantissa; if **Digits** is less than 15, 53 bits are used, if **Digits** is strictly greater than 15, the number of bits is a roundoff of **Digits** times $\log_2(10)$.

An expression can be coerced into a floating number with the `evalf` command (see Section 6.8.1 p.171). The `evalf` command may have an optional second argument which will specify the precision to use.

Note that if an expression contains a floating number, evaluation will try to convert other arguments to floating point numbers in order to coerce the whole expression to a single floating number.

Examples.

- *Input:*

Output:

$$\frac{3}{2}$$

• *Input:*

$$1.0+1/2$$

Output:

$$1.5$$

• *Input:*

$$\exp(\pi \cdot \sqrt{20})$$

Output:

$$e^{2\pi\sqrt{5}}$$

With evalf, input:

$$\text{evalf}(\exp(\pi \cdot 2 \cdot \sqrt{5}))$$

Output:

$$1263794.75367$$

• *Input:*

$$1.1^{20}$$

Output:

$$6.72749994932$$

• *Input:*

$$\sqrt{2}^{21}$$

Output:

$$\sqrt{2} \cdot 2^{10}$$

• *Input (for a result with 30 digits):*

$$\text{Digits}=30$$

Input (for the numeric value of $e^{\pi\sqrt{163}}$):

$$\text{evalf}(\exp(\pi \cdot \sqrt{163}))$$

Output:

$$0.262537412640768743999999999985 \times 10^8$$

Note that `Digits` is now set to 30. If you didn't want to change the value of `Digits`, you could have entered:

Input:

$$\text{evalf}(\exp(\pi \cdot \sqrt{163}), 30)$$

6.8.2 The standard infix operators on real numbers: + - * / ^

The +, -, *, /, and ^ operators are the usual infix operators to do addition, subtraction, multiplication, division and raising to a power.

Examples.

- *Input:*

3+2

Output:

5

- *Input:*

3-2

Output:

1

- *Input:*

3*2

Output:

6

- *Input:*

3/2

Output:

$\frac{3}{2}$

- *Input:*

3.2/2.1

Output:

1.52380952381

- *Input:*

3^2

Output:

9

- *Input:*

3.2^2.1

Output:

11.5031015682

Remark.

You can use the square key or the cube key if your keyboard has one; for example: 3^2 returns 9.

Remarks on non integral powers.

If x is not an integer, then $a^x = \exp(x \ln(a))$, hence if x is not rational, then a^x is well-defined only for $a > 0$. If x is rational and $a < 0$, the principal branch of the logarithm is used, leading to a complex number. Note the difference between $\sqrt[n]{a}$ and $a^{\frac{1}{n}}$ when n is an odd integer.

Example.

To draw the graph of $y = \sqrt[3]{x^3 - x^2}$:

Input:

```
plotfunc(ifte(x>0,(x^3-x^2)^(1/3),
-(x^2-x^3)^(1/3)),x,xstep=0.01)
```

You might also input:

```
plotimplicit(y^3=x^3-x^2)
```

but this is much slower and much less accurate.

6.8.3 Prefixed division on reals: rdiv

The **rdiv** command is the prefixed form of the usual division operator.

Examples.

- *Input:*

```
rdiv(3,2)
```

Output:

$$\frac{3}{2}$$

- *Input:*

```
rdiv(3.2,2.1)
```

Output:

1.52380952381

6.8.4 n -th root: root

The `root` command finds roots of numbers.

- `root` takes two arguments:
 n and a , numbers.
- `root(n, a)` returns the n th root of a (i.e. $a^{1/n}$). If $a < 0$, the n -th root is a complex number with argument $2\pi/n$.

Examples.

- *Input:*

```
root(3,2)
```

Output:

$$2^{\frac{1}{3}}$$

- *Input:*

```
root(3,2.0)
```

Output:

$$1.25992104989$$

- *Input:*

```
root(3,sqrt(2))
```

Output:

$$2^{\frac{1}{6}}$$
6.8.5 The exponential integral function: Ei

The exponential integral `Ei` is defined for non-zero real numbers x by

$$\text{Ei}(x) = \int_{t=-\infty}^x \frac{\exp(t)}{t} dt.$$

For $x > 0$, this integral is improper but the principal value exists. This function satisfies $\text{Ei}(0) = -\infty$, $\text{Ei}(-\infty) = 0$.

Since

$$\frac{\exp(x)}{x} = \frac{1}{x} + 1 + \frac{x}{2!} + \frac{x^2}{3!} + \dots,$$

the `Ei` function can be extended to $\mathbb{C} - \{0\}$ (with a branch cut on the positive real axis) by

$$\text{Ei}(z) = \ln(z) + \gamma + x + \frac{x^2}{2 \cdot 2!} + \frac{x^3}{3 \cdot 3!} + \dots$$

where $\gamma = 0.57721566490\dots$ is Euler's constant.

The `Ei` command takes one or two arguments.

With one argument, the `Ei` command computes the exponential integral.

- Ei takes one argument:
 z , a complex number.
- Ei(z) returns the value of the exponential integral at z .

Examples.

- *Input:*

Ei(1.0)

Output:

1.89511781636

- *Input:*

Ei(-1.0)

Output:

-0.219383934396

- *Input:*

Ei(1.)-Ei(-1.)

Output:

2.11450175075

- *Input:*

int((exp(x)-1)/x,x=-1..1.)

Output:

2.11450175075

- The input:

Input:

evalf(Ei(-1)-sum((-1)^n/n/n!,n=1..100))

approximates the Euler's constant γ

Output:

0.577215664902

Another type of exponential integral is

$$E_1(x) = \int_x^\infty \frac{\exp(-t)}{t} dt = \int_1^\infty \frac{\exp(-tx)}{t} dt$$

which satisfies

$$E_1(x) = -\text{Ei}(-x)$$

This can be generalized to

$$E_n(x) = \int_1^\infty \frac{\exp(-tx)^n}{t} dt$$

These functions satisfy

$$\begin{aligned} E_1(x) &= -\text{Ei}(x) \\ E_2(x) &= e^{-x} + x\text{Ei}(-x) = e^{-x} - x * E_1(x) \end{aligned}$$

and, for $n \geq 2$,

$$E_n(x) = (e^{-x} - xE_{n-1}(x))/(n-1)$$

With two arguments, the **Ei** command computes this version of the exponential integral.

- **Ei** takes two arguments:
 - z , a complex number.
 - n , a positive integer.
- **Ei**(z, n) returns the value of $E_n(z)$.

Examples.

- *Input:*

Ei(1.0,1)

Output:

0.219383934396

- *Input:*

Ei(3.0,2)

Output:

0.0106419250853

6.8.6 The logarithmic integral function: **Li**

The logarithmic integral function is defined by

$$\text{Li}(x) = \text{Ei}(\ln(x)) = \int_{t=0}^{\exp(x)} \frac{1}{\ln(t)} dt$$

The **Li** command computes the logarithmic integral.

- **Li** takes one argument:
 - z , a complex number.
- **Li**(z) returns the value of the logarithmic integral $\text{Li}(z)$.

Example.

Input:

Li(2.0)

Output:

1.04516378012

6.8.7 The cosine integral function: Ci

The cosine integral function is defined by

$$\begin{aligned}\text{Ci}(x) &= \int_{+\infty}^x \frac{\cos(t)}{t} dt \\ &= \ln(t) + \gamma + \int_{t=0}^x \frac{\cos(t) - 1}{t} dt\end{aligned}$$

and $\text{Ci}(0) = -\infty$, $\text{Ci}(-\infty) = i\pi$ and $\text{Ci}(+\infty) = 0$.

The **Ci** command computes the cosine integral function.

- **Ci** takes one argument:
 z , a complex number.
- **Ci**(z) returns the value of the cosine integral function $\text{Ci}(z)$.

Examples.

- *Input:*

Ci(1.0)

Output:

0.337403922901

- *Input:*

Ci(-1.0)

Output:

0.337403922901 + 3.14159265359i

- *Input:*

Ci(1.0) - **Ci**(-1.0)

Output:

-3.14159265359i

6.8.8 The sine integral function: Si

The sine integral function is defined by

$$\text{Si}(x) = \int_0^x \frac{\sin(t)}{t} dt$$

and $\text{Si}(0) = 0$, $\text{Si}(-\infty) = -\pi/2$ and $\text{Si}(+\infty) = \pi/2$. Note that *Si* is an odd function.

The **Si** command computes the sine integral function.

- **Si** command takes one argument:
 z , a complex number.

- `Si(z)` returns the value of the sine integral function $\text{Si}(z)$.

Example.*Input:*`Si(1.0)`*Output:*

0.946083070367

Input:`Si(-1.0)`*Output:*

-0.946083070367

6.8.9 The Heaviside function: Heaviside

The Heaviside function is the step function

$$H(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$$

The `Heaviside` command computes the Heaviside function.

- `Heaviside` takes one argument:
 x , a real number.
- `Heaviside(x)` returns the value of the Heaviside function $H(x)$.

Examples.

- *Input:*

`Heaviside(2)`*Output:*

1

- *Input:*

`Heaviside(-4)`*Output:*

0

6.8.10 The Dirac distribution: Dirac

The Dirac δ distribution is the distributional derivative of the Heaviside function. This means that

$$\int_{-\infty}^{\infty} \delta(x) dx = 1$$

and, in fact,

$$\int_a^b \delta(x) dx = \begin{cases} 1 & \text{if } 0 \in [a, b] \\ 0 & \text{otherwise} \end{cases}$$

The defining property of the Dirac distribution is that

$$\int_{-\infty}^{\infty} \delta(x) f(x) dx = f(0)$$

and consequently

$$\int_a^b \delta(x - c) f(x) dx = f(c)$$

as long as c is in $[a, b]$.

The **Dirac** command represents the Dirac distribution.

Examples.

- *Input:*

```
int(Dirac(x)*sin(x),x,-1,2)
```

Output:

$\sin(0)$

- *Input:*

```
int(Dirac(x-1)*sin(x),x,-1,2)
```

Output:

$\sin(1)$

If you have Dirac compute a value:

- **Dirac** it takes one argument:
 x , a real number.
- **Dirac**(x) returns ∞ if $x = 0$, it returns 0 otherwise.

6.8.11 Error function: erf

The error function `erf` is defined by:

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

where the constant $\frac{2}{\sqrt{\pi}}$ is chosen so that

$$\operatorname{erf}(+\infty) = 1, \quad \operatorname{erf}(-\infty) = -1$$

since

$$\int_0^{+\infty} e^{-t^2} dt = \frac{\sqrt{\pi}}{2}$$

The `erf` command computes the error function.

- `erf` takes one argument:
 a , a number.
- `erf(a)` returns the value of $\operatorname{erf}(a)$.

Examples.

- *Input:*

```
erf(1)
```

Output:

```
erf(1)
```

- *Input:*

```
erf(1.0)
```

Output:

```
0.84270079295
```

- *Input:*

```
erf(1/(sqrt(2)))*1/2+0.5
```

Output:

```
0.841344746069
```

Remark.

The relation between `erf` and `normal_cdf` (see Section 9.4.7 p.768) is:

$$\operatorname{normal_cdf}(x) = \frac{1}{2} + \frac{1}{2} \operatorname{erf}\left(\frac{x}{\sqrt{2}}\right)$$

Indeed, making the change of variable $t = u * \sqrt{2}$ in

$$\operatorname{normal_cdf}(x) = \frac{1}{2} + \frac{1}{\sqrt{2\pi}} \int_0^x e^{-t^2/2} dt$$

gives:

$$\text{normal_cdf}(x) = \frac{1}{2} + \frac{1}{\sqrt{\pi}} \int_0^{\frac{x}{\sqrt{2}}} e^{-u^2} du = \frac{1}{2} + \frac{1}{2} \text{erf}\left(\frac{x}{\sqrt{2}}\right)$$

Check:

Input:

```
normal_cdf(1.0)
```

Output:

```
0.841344746069
```

6.8.12 Complementary error function: `erfc`

The complementary error function is defined by

$$\text{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^{+\infty} e^{-t^2} dt = 1 - \text{erf}(x)$$

Hence $\text{erfc}(0) = 1$, since

$$\int_0^{+\infty} e^{-t^2} dt = \frac{\sqrt{\pi}}{2}$$

The `erfc` command computes the complementary error function.

- `erfc` takes one argument:
 a , a number.
- `erfc(a)` returns the value of the complementary error function $\text{erfc}(a)$.

Examples.

- *Input:*

```
erfc(1)
```

Output:

```
1 - erf(1)
```

- *Input:*

```
1 - erfc(1/(sqrt(2)))*0.5
```

Output:

```
0.841344746069
```

Remark.

The relation between `erfc` and `normal_cdf` (see Section 9.4.7 p.768) is:

$$\text{normal_cdf}(x) = 1 - \frac{1}{2} \text{erfc}\left(\frac{x}{\sqrt{2}}\right)$$

Check:

Input:

```
normal_cdf(1.0)
```

Output:

```
0.841344746069
```

6.8.13 The Γ function: Gamma

The Gamma function is defined by

$$\Gamma(x) = \int_0^{+\infty} e^{-t} t^{x-1} dt, \text{ if } x > 0$$

If x is a positive integer, Γ is computed by applying the recurrence:

$$\Gamma(x+1) = x * \Gamma(x), \quad \Gamma(1) = 1$$

Hence:

$$\Gamma(n+1) = n!$$

and the Gamma function is used to generalize the factorial (see Section 6.6.1 p.158).

The `Gamma` command computes the Gamma function.

- `Gamma` takes one argument:
 a , a number.
- `Gamma(a)` returns the value $\Gamma(a)$.

Examples.

- *Input:*

```
Gamma(5)
```

Output:

```
24
```

- *Input:*

```
Gamma(0.7)
```

Output:

```
1.29805533265
```

- *Input:*

```
Gamma(-0.3)
```

Output:

```
-4.32685110883
```

Indeed: `Gamma(0.7)=-0.3*Gamma(-0.3)`

- *Input:*

```
Gamma(-1.3)
```

Output:

```
3.32834700679
```

Indeed `Gamma(0.7)=-0.3*Gamma(-0.3)=(-0.3)*(-1.3)*Gamma(-1.3)`

6.8.14 The upper incomplete γ function: `ugamma`

The upper incomplete γ function is defined by

$$\Gamma(a, b) = \int_b^{+\infty} e^{-t} t^{a-1} dt.$$

The `ugamma` command computes the upper incomplete γ function.

- `ugamma` takes two arguments:
 - a , a number.
 - b , a positive real number.
- `ugamma(a, b)` returns the value of $\Gamma(a, b)$.

Examples.

- *Input:*

```
ugamma(3.0, 2.0)
```

Output:

```
1.35335283237
```

- *Input:*

```
ugamma(-1.3, 2)
```

Output:

```
0.0142127568837
```

6.8.15 The lower incomplete γ function: `igamma`

The lower incomplete γ function is defined by

$$\gamma(a, b) = \int_0^b e^{-t} t^{a-1} dt.$$

The `igamma` command computes the lower incomplete γ function.

- `igamma` takes two mandatory arguments and one optional argument:
 - a , a number.
 - b , a positive real number.
 - Optionally, the number 1.
- `igamma(a, b)` returns $\gamma(a, b)$.
- `igamma($a, b, 1$)` returns a normalized version of the function; namely $\gamma(a, b)/\Gamma(a)$.

Examples.

- *Input:*

```
igamma(4.0,3.0)
```

Output:

```
2.11660866731
```

- *Input:*

```
igamma(4.0,3.0,1)
```

Output:

```
0.352768111218
```

since $\Gamma(4) = 6$ and $2.11660866731/6 = 0.352768111218$.

6.8.16 The β function: Beta

The β function is defined by

$$\beta(x, y) = \int_0^1 t^{x-1} (1-t)^{y-1} dt = \frac{\Gamma(x) * \Gamma(y)}{\Gamma(x+y)}$$

This is defined for x and y positive reals (to ensure the convergence of the integral) and by extension for x and y if they are not negative integers.

Remarkable values:

$$\beta(1, 1) = 1, \quad \beta(n, 1) = \frac{1}{n}, \quad \beta(n, 2) = \frac{1}{n(n+1)}$$

The **Beta** command computes the β function.

- **Beta** takes two arguments:
 a and b , real numbers.
- **Beta**(a, b) returns the value of the $\beta(a, b)$.

Examples.

- *Input:*

```
Beta(5,2)
```

Output:

```
 $\frac{1}{30}$ 
```

- *Input:*

```
Beta(x,y)
```

Output:

$$\frac{\Gamma(x)\Gamma(y)}{\Gamma(x+y)}$$

- *Input:*

`Beta(5.1, 2.2)`

Output:

0.0242053671402

6.8.17 Derivatives of the DiGamma function: Psi

The DiGamma function is the derivative of the logarithm of the Γ function (see Section 6.8.13 p.183),

$$\psi(z) = \frac{d}{dz} \ln(\Gamma(z)) = \frac{\Gamma'(z)}{\Gamma(z)}$$

This function is used to evaluate sums of rational functions having poles at integers.

The **Psi** function computes the DiGamma function and its derivatives.

- **Psi** takes one mandatory argument and one optional argument:
 - a , a real number.
 - Optionally, n , a non-negative integer.
- **Psi**(a) returns the value of the DiGamma function $\psi(a)$.
- **Psi**(a, n) returns the n th derivative of the DiGamma function at $x = a$.

Examples.

- *Input:*

`Psi(3)`

Output:

$$\frac{3}{2} - \gamma$$

- *Input:*

`evalf(Psi(3))`

Output:

0.922784335098

- *Input:*

`Psi(3, 1)`

Output:

$$\frac{\pi^2}{6} - \frac{5}{4}$$

6.8.18 The ζ function: Zeta

The ζ function is defined by

$$\zeta(x) = \sum_{n=1}^{+\infty} \frac{1}{n^x}$$

for $x > 1$, and by its meromorphic continuation for $x < 1$.

The **Zeta** command computes the ζ function.

- **Zeta** takes one argument:
 x , a real number.
- **Zeta**(x) returns the value of the ζ function $\zeta(x)$.

Examples.

- *Input:*

Zeta(2)

Output:

$$\frac{\pi^2}{6}$$

- *Input:*

Zeta(4)

Output:

$$\frac{\pi^4}{90}$$

6.8.19 Airy functions: Airy_Ai and Airy_Bi

The Airy functions of the first and second kind are defined by

$$\begin{aligned} \text{Ai}(x) &= (1/\pi) \int_0^\infty \cos(t^3/3 + x * t) dt \\ \text{Bi}(x) &= (1/\pi) \int_0^\infty (e^{-t^3/3} + \sin(t^3/3 + x * t)) dt \end{aligned}$$

They have the properties that, if f and g are two entire series solutions of

$$w'' - x * w = 0$$

then

$$\begin{aligned} \text{Ai}(x) &= \text{Ai}(0) * f(x) + \text{Ai}'(0) * g(x) \\ \text{Bi}(x) &= \sqrt{3}(\text{Ai}(0) * f(x) - \text{Ai}'(0) * g(x)) \end{aligned}$$

more precisely:

$$\begin{aligned} f(x) &= \sum_{k=0}^{\infty} 3^k \left(\frac{\Gamma(k + \frac{1}{3})}{\Gamma(\frac{1}{3})} \right) \frac{x^{3k}}{(3k)!} \\ g(x) &= \sum_{k=0}^{\infty} 3^k \left(\frac{\Gamma(k + \frac{2}{3})}{\Gamma(\frac{2}{3})} \right) \frac{x^{3k+1}}{(3k+1)!} \end{aligned}$$

The `Airy_Ai` and `Airy_Bi` commands compute the Airy functions.

- `Airy_Ai` and `Airy_Bi` take one argument:
 x , a real number.
- `Airy_Ai(x)` and `Airy_Bi(x)` return the values of the Airy functions.

Examples.

- *Input:*

`Airy_Ai(1)`

Output:

0.135292416313

- *Input:*

`Airy_Bi(1)`

Output:

1.20742359495

- *Input:*

`Airy_Ai(0)`

Output:

0.355028053888

- *Input:*

`Airy_Bi(0)`

Output:

0.614926627446

6.9 Permutations

A *permutation* p of size n is a bijection from $[0..n-1]$ to $[0..n-1]$ and is represented by the list: $[p(0), p(1), p(2) \dots p(n-1)]$.

For example, the permutation p represented by $[1, 3, 2, 0]$ is the function from $[0, 1, 2, 3]$ to $[0, 1, 2, 3]$ defined by:

$$p(0) = 1, \quad p(1) = 3, \quad p(2) = 2, \quad p(3) = 0$$

A *cycle* c of size p , represented by the list $[a_0, \dots, a_{p-1}]$ ($0 \leq a_k \leq n-1$), is the permutation such that

$$c(a_i) = a_{i+1} \text{ for } (i = 0..p-2), \quad c(a_{p-1}) = a_0, \quad c(k) = k \text{ otherwise}$$

For example, the cycle c represented by the list $[3, 2, 1]$ is the permutation c defined by $c(3) = 2$, $c(2) = 1$, $c(1) = 3$, $c(0) = 0$ (i.e. the permutation represented by the list $[0, 3, 1, 2]$).

6.9.1 Random permutation: randperm shuffle

The **randperm** command computes a random permutation. **shuffle** is a synonym for **randperm**.

- **randperm** takes one argument:
 n , an integer.
- **randperm**(n) returns a random permutation of $[0..n-1]$.

Example.

Input:

```
randperm(3)
```

Output (example):

```
[2, 0, 1]
```

6.9.2 Previous and next permutation: prevperm nextperm

The set of n -tuples of an ordered set can be put in *lexicographic order*, where the tuple (a_1, a_2, \dots, a_n) comes before (b_1, b_2, \dots, b_n) exactly when for some k (possibly $k = 0$), $a_i = b_i$ for $i = 1, \dots, k-1$ and $a_k < b_k$. For example, the set of permutations of size 3 in lexicographic order is

```
(0, 1, 2)
(0, 2, 1)
(1, 0, 2)
(1, 2, 0)
(2, 0, 1)
(2, 1, 0)
```

The **prevperm** and **nextperm** commands find the preceding and succeeding permutation.

- **prevperm** takes one argument:
 p , a permutation.
- **prevperm(p)** returns the previous permutation in lexicographic order, or **undef** if there is no previous permutation.

Example.*Input:*`prevperm([0,3,1,2])`*Output:*`[0,2,3,1]`

- **nextperm** takes one argument:
 p , a permutation.
- **nextperm(p)** returns the next permutation in lexicographic order, or **undef** if there is no next permutation.

Example.*Input:*`nextperm([0,2,3,1])`*Output:*`[0,3,1,2]`**6.9.3 Decomposing a permutation into a product of disjoint cycles: `permu2cycles`**

Any permutation can be decomposed as a sequence of cycles which have no elements in common. For example, the permutation `[1,3,4,0,2]` can be written as a combination of the cycles `[0,1,3]` and `[2,4]`.

The **permu2cycles** command decomposes a permutation into a combination of cycles.

- **permu2cycles** takes one argument:
 p , a permutation.
- **permu2cycles(p)** returns the decomposition of p as a product of disjoint cycles. A cycle is represented by a list, a cyclic decomposition is represented by a list of lists.

Examples.

- *Input:*

`permu2cycles([1,3,4,5,2,0])`

Output:

$[[0, 1, 3, 5], [2, 4]]$

In the answer the cycles of size 1 are omitted, except if $n - 1$ is a fixed point of the permutation (this is required to find the value of n from the cycle decomposition).

• *Input:*

`permu2cycles([0,1,2,4,3,5])`

Output:

$[[5], [3, 4]]$

• *Input:*

`permu2cycles([0,1,2,3,5,4])`

Output:

$[[4, 5]]$

6.9.4 Product of cycles to permutation: `cycles2permu`

The `cycles2permu` command is the inverse of `permu2cycles`; it turns a sequence of cycles into a permutation.

- `cycles2permu` takes one argument:
 c , a list of cycles.
- `cycles2permu(c)` returns the permutation (of size n chosen as small as possible) that is the product of the given cycles.

Examples.

• *Input:*

`cycles2permu([[1,3,5],[2,4]])`

Output:

$[0, 3, 4, 5, 2, 1]$

• *Input:*

`cycles2permu([[2,4]])`

Output:

$[0, 1, 4, 3, 2]$

• *Input:*

`cycles2permu([[5],[2,4]])`

Output:

$[0, 1, 4, 3, 2, 5]$

6.9.5 Transforming a cycle into a permutation: `cycle2perm`

A cycle is a type of permutation, but has a different representation.

The `cycle2perm` command converts a cycle to the cycle written as a permutation.

- `cycle2perm` takes one argument:
 c , a cycle c .
- `cycle2perm(c)` returns the permutation of size n corresponding to the cycle c , where n is chosen as small as possible (see also `permu2cycles` and `cycles2permu`).

Example.

Input:

```
cycle2perm([1,3,5])
```

Output:

```
[0,3,2,5,4,1]
```

6.9.6 Transforming a permutation into a matrix: `permu2mat`

The matrix of a permutation p of size n is the matrix obtained by permuting the rows of the identity matrix of size n with the permutation p . Multiplying this matrix by a column vector of size n is the same as permuting the elements of the vector with the permutation p .

The `permu2mat` command finds the matrix of a given permutation.

- `permu2mat` takes one argument:
 p , a permutation p .
- `permu2mat(p)` returns the matrix of the permutation p .

Example.

Input:

```
permu2mat([2,0,1])
```

Output:

$$\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

6.9.7 Checking for a permutation: `is_permu`

A permutation can be written as a list, but not every list corresponds to a permutation. The `is_permu` is a boolean function which checks to see if a given list is a permutation.

- `is_permu` takes one argument:
 L , a list.

- `is_permu(L)` returns 1 if L is a permutation and returns 0 if L is not a permutation.

Examples.

- *Input:*

```
is_permu([2,1,3])
```

Output:

0

- *Input:*

```
is_permu([2,1,3,0])
```

Output:

1

6.9.8 Checking for a cycle: `is_cycle`

The `is_cycle` command is a boolean function which checks to see if a list represents a cycle.

- `is_cycle` takes one argument:
 L , a list.
- `is_cycle(L)` returns 1 if L is a cycle and returns 0 if L is not a cycle.

Examples.

- *Input:*

```
is_cycle([2,1,3])
```

Output:

1

- *Input:*

```
is_cycle([2,1,3,2])
```

Output:

0

6.9.9 Product of two permutations: `p1op2` `c1op2` `p1oc2` `c1oc2`

Permutations are functions, and so can be composed. Since cycles can be represented differently than other permutations, there are commands for composing permutations of different types.

Warning.

Composition is done using the standard mathematical notation; that is, the function given as the second argument is performed first.

The `p1op2` command composes two permutations.

- `p1op2` takes two arguments:
 p_1 and p_2 , permutations.
- `p1op2(p_1, p_2)` returns the permutation $p_1 \circ p_2$ obtained by composition.

Example.

Input:

```
p1op2([3,4,5,2,0,1],[2,0,1,4,3,5])
```

Output:

```
[5,3,4,0,2,1]
```

The `c1op2` command composes a cycle and a permutation.

- `c1op2` takes two arguments:
 - c_1 , a cycle.
 - p_2 , a permutation.
- `c1op2(c_1, p_2)` returns the permutation $c_1 \circ p_2$ obtained by composition.

Example.

Input:

```
c1op2([3,4,5],[2,0,1,4,3,5])
```

Output:

```
[2,0,1,5,4,3]
```

The `p1oc2` command composes a permutation and a cycle.

- `p1oc2` takes two arguments:
 - p_1 , a permutation.
 - c_2 , a cycle.
- `p1oc2(p_1, c_2)` returns the permutation $p_1 \circ c_2$ obtained by composition.

Example.

Input:

```
p1oc2([3,4,5,2,0,1],[2,0,1])
```

Output:

`[4, 5, 3, 2, 0, 1]`

The `c1oc2` command composes two cycles.

- `c1oc2` takes two arguments:
 c_1 and c_2 , cycles.
- `c1oc2(c_1, c_2)` returns the permutation $c_1 \circ c_2$ obtained by composition.

Example.

Input:

`c1oc2([3, 4, 5], [2, 0, 1])`

Output:

`[1, 2, 0, 4, 5, 3]`

6.9.10 Signature of a permutation: `signature`

Every permutation can be decomposed into a product of transpositions (cycles with only two elements). The number of transpositions is not unique, but for any permutation the number will be either odd or even. The signature of a permutation is equal to:

- 1 if the permutation is equal to an even product of transpositions,
- -1 if the permutation is equal to an odd product of transpositions.

The signature of a cycle of size k is: $(-1)^{k+1}$.

The `signature` command computes the signature of a permutation.

- `signature` takes one argument:
 p , a permutation.
- `signature(p)` returns the signature of the permutation p .

Example.

Input:

`signature([3, 4, 5, 2, 0, 1])`

Output:

`-1`

6.9.11 Inverse of a permutation: `perminv`

Every permutation has an inverse, which is also a permutation.

The `perminv` command computes the inverse of a permutation.

- `perminv` takes one argument:
 p , a permutation.
- `perminv(p)` returns the permutation that is the inverse of p .

Example.*Input:*`perminv([1,2,0])`*Output:*`[2,0,1]`**6.9.12 Inverse of a cycle: cycleinv**

The inverse of a cycle will be another cycle.

The `cycleinv` command computes the inverse of a cycle.

- `cycleinv` takes one argument:
 c , a cycle.
- `cycleinv(c)` returns the cycle that is the inverse of c .

Example.*Input:*`cycleinv([2,0,1])`*Output:*`[1,0,2]`**6.9.13 Order of a permutation: permuorder**

If any permutation p on a finite set $[0, \dots, n-1]$ is repeated often enough, it reach be the identity permutation. The smallest m such that p^m is the identity is called the *order* of p .

The `permuorder` command computes the order of a permutation.

- `permuorder` takes one argument:
 p , a permutation.
- `permuorder(p)` returns the order of the permutation p .

Examples.

- *Input:*

`permuorder([0,2,1])`*Output:*`2`

- *Input:*

`permuorder([3,2,1,4,0])`*Output:*`6`

6.9.14 The group generated by two permutations: groupermu

Given permutations a and b , the group they generate is the set of all possible compositions of any number of a s and any number of b s.

The `groupermu` command computes the group generated by two permutations.

- `groupermu` takes two arguments:
 a and b , permutations.
- `groupermu(a,b)` returns the group of the permutations generated by a and b .

Example.

Input:

```
groupermu([0,2,1,3],[3,1,2,0])
```

Output:

$$\begin{bmatrix} 0 & 2 & 1 & 3 \\ 3 & 1 & 2 & 0 \\ 0 & 1 & 2 & 3 \\ 3 & 2 & 1 & 0 \end{bmatrix}$$

6.10 Complex numbers

Note that complex numbers, as well as being numbers, are used to represent points in the plane (see Section 13.6.2 p.886). Some functions and operators which work on complex numbers also work on points.

6.10.1 The usual complex operators: + - * / ^

The `+`, `-`, `*`, `/`, `^` operators are the usual operators to perform addition, subtraction, multiplication, division and for raising to a power.

Input:

```
(1+2*i)^2
```

Output:

```
-3 + 4i
```

6.10.2 The real and imaginary parts of a complex number: re real im imag

The `re` (or `real`) and `im` (or `imag`) commands find the real and imaginary parts of a complex number.

The `re` command finds the real part of a complex number.
`real` is a synonym for `re`.

- `re` takes one argument:
 a , a complex number (or point).

- **re**(a) returns the real part of the complex number a (or the projection of the point a onto the x axis).

Example.*Input:*

```
re(3+4*i)
```

Output:

3

The **im** command finds the imaginary part of a complex number. **imag** is a synonym for **im**.

- **im** takes one argument:
 a , a complex number (or point).
- **im**(a) returns the imaginary part of the complex number a (or the projection of the point a onto the y axis).

Example.*Input:*

```
im(3+4*i)
```

Output:

4

6.10.3 Writing a complex number z in rectangular form: **evalc**

The **evalc** command will ensure that a complex number is in rectangular form.

- **evalc** takes one argument:
 z , a complex number.
- **evalc**(z) returns z written as **re**(z)+**i*****im**(z).

Example.*Input:*

```
evalc(sqrt(2)*exp(i*pi/4))
```

Output:

1 + i

6.10.4 The modulus and argument of a complex number: `abs` `arg`

A complex number z can be written in polar form $re^{i\theta}$, where r is the modulus and θ is the argument. The angle θ is only determined up to a multiple of 2π ; there will be a unique value in the interval $(-\pi, \pi]$, the value in this interval is called the *principal value* of the argument.

The `abs` and `arg` commands find the modulus and argument of a complex number.

The `abs` command finds the modulus of a complex number (see also Section 6.16.2 p.246).

- `abs` takes one argument:
 z , a complex number.
- `abs(z)` returns the modulus $|z|$.

Example.

Input:

```
abs(3+4*i)
```

Output:

```
5
```

The `arg` command finds the argument of a complex number.

- `arg` takes one argument:
 z , a complex number.
- `arg(z)` returns the principal value of the argument of z .

Examples.

- *Input:*

```
arg(3+4*i)
```

Output:

```
arctan(4/3)
```

- *Input:*

```
arg(3.0+4.0*i)
```

Output:

```
0.927295218002
```

6.10.5 The normalized complex number: `normalize unitV`

The `normalize` command finds the unit complex number with the same direction as a given complex number.

`unitV` is a synonym for `normalize`.

- `normalize` takes one argument:
 z , a non-zero complex number.
- `normalize(z)` returns the unit complex number with the same direction as z , namely z divided by the modulus of z .

Example.

Input:

```
normalize(3+4*i)
```

Output:

$$\frac{3 + 4i}{5}$$

6.10.6 Conjugate of a complex number: `conj`

The `conj` command finds the conjugate of a complex number.

- `conj` takes one argument:
 z , a complex number.
- `conj(z)` returns the complex conjugate of z .

Example.

Input:

```
conj(3+4*i)
```

Output:

$$3 - 4i$$

6.10.7 Multiplication by the complex conjugate: `mult_c_conjugate`

The denominator of a complex expression can be made a real number by multiplying the numerator and denominator of the expression by the complex conjugate of the denominator. The `mult_c_conjugate` can perform this multiplication.

- `mult_c_conjugate` takes one argument:
 $expr$, a complex expression.
- `mult_c_conjugate($expr$)` returns the following:
 - If $expr$ is a fraction with a complex (non-real) denominator, then this expression is returned with the numerator and denominator multiplied by the complex conjugate of the denominator.

- If *expr* is a fraction with a real denominator (if *expr* is not a fraction, it is regarded as a fraction with a denominator of 1), then this expression is returned with the numerator and denominator multiplied by the complex conjugate of the numerator.

Examples.

- *Input:*

```
mult_c_conjugate((2+i)/(2+3*i))
```

Output:

$$\frac{(2+i)(2-3i)}{(2+3i)(2-3i)}$$

- *Input:*

```
mult_c_conjugate((2+i)/2)
```

Output:

$$\frac{(2+i)(2-i)}{2(2-i)}$$

6.10.8 Barycenter of complex numbers: barycenter

The *barycenter*, or center of mass, of a set of points A_1, A_2, \dots, A_n with masses $\alpha_1, \alpha_2, \dots, \alpha_n$ is

$$\frac{\alpha_1 A_1 + \dots + \alpha_n A_n}{\alpha_1 + \dots + \alpha_n}$$

This formula makes sense even if the α_j are not positive real numbers, and is still called the barycenter of the weighted points.

The **barycenter** command computes the barycenter of a set of weighted points.

- **barycenter** takes an unspecified number of arguments:
each argument is a list $l_j = [A_j, \alpha_j]$ containing a point A_j (or the affix of a point) and a weight α_j for the point. The sum of the weights needs to be non-zero.
These lists can also be given as two columns of a matrix.
- **barycenter**(l_1, l_2, \dots, l_n) returns the barycenter of the points A_j weighted by the real coefficients α_j . If $\sum \alpha_j = 0$, **barycenter** returns an error.

Warning.

The **barycenter** command returns a point, not a complex number. To have a complex number in the output, the input must be **affix(barycenter(l_1, l_2))** (see Section 13.13.1 p.933).

Example.

Input:

```
affix(barycenter([1+i,2],[1-i,1]))
```

or:

```
affix(barycenter([[1+i,2],[1-i,1]]))
```

Output:

$$\frac{3+i}{3}$$

6.11 Algebraic numbers

6.11.1 Definition

A real algebraic number is a real root of a polynomial with integer coefficients.

A complex algebraic number is a root of a polynomial with coefficients which are Gaussian integers.

6.11.2 Minimum polynomial of an algebraic number: `pmin`

The minimal polynomial of an algebraic number is the monic polynomial of smallest degree with integer coefficients which has the algebraic number as a root.

The `pmin` command finds the minimum polynomial of an algebraic number.

- `pmin` takes one mandatory argument and one optional argument:
 - α , an algebraic number.
 - Optionally, x , a variable name to use as the variable in the polynomial.
- `pmin(α)` returns the minimal polynomial for α , where the polynomial is given as a list of the coefficients (see Section 6.27.1 p.351).
`pmin(α , x)` returns the minimal polynomial for α as a symbolic expression with the variable x .

Examples.

- *Input:*

```
pmin(sqrt(2) + sqrt(3))
```

Output:

$$\llbracket 1, 0, -10, 0, 1 \rrbracket$$

- *Input:*

```
pmin(sqrt(2) + sqrt(3), x)
```

Output:

$$x^4 - 10x^2 + 1$$

Note that $(\sqrt{2} + \sqrt{3})^2 = 5 + 2\sqrt{6}$ and so $((\sqrt{2} + \sqrt{3})^2 - 5)^2 = 24$, which can be rewritten as $(\sqrt{2} + \sqrt{3})^4 - 10(\sqrt{2} + \sqrt{3})^2 + 1 = 0$.

• *Input:*

```
pmin(sqrt(2) + i*sqrt(3))
```

Output:

$$\llbracket 1, 0, 2, 0, 25 \rrbracket$$

• *Input:*

```
pmin(sqrt(2) + i*sqrt(3), z)
```

Output:

$$z^4 + 2z^2 + 25$$

• *Input:*

```
pmin(sqrt(2) + 2*i)
```

Output:

$$\llbracket 1, 0, 4, 0, 36 \rrbracket$$

• *Input:*

```
pmin(sqrt(2) + 2*i, z)
```

Output:

$$z^4 + 4z^2 + 36$$

6.12 Algebraic expressions

6.12.1 Evaluating an expression: eval

The **eval** command is used to evaluate an expression. Since **Xcas** always evaluates expressions entered in the command line, **eval** is mainly used to evaluate a sub-expression in the expression editor (see Section 4.3 p.88).

Examples.

• *Input:*

```
a:=2
```

Output:

2

• *Input:*

```
eval(2+3*a)
```

or:

```
2+3*a
```

Output:

```
8
```

6.12.2 Changing the evaluation level: eval_level

When it evaluates expressions, the maximum number of recursions that **Xcas** will do it called the *evaluation level*. This is 25 by default, but you can change the default level with the **eval** box in the CAS configuration screen (see section 3.5.7).

The **eval_level** command will change the evaluation level for the current session.

- **eval_level** takes one optional argument:
Optionally n , a positive integer.
- **eval_level()** returns the current evaluation level.
- **eval_level(n)** sets the evaluation level to n .

Example.

Input:

```
purge(a,b,c)
a:=b+1; b:=c+1; c:=3;
```

Input:

```
eval_level()
```

Output:

```
25
```

Input:

```
a,b,c
```

Output:

```
5,4,3
```

Input:

```
eval_level(1)
a,b,c
```

Output:

```
b + 1, c + 1, 3
```

Input:

```
eval_level(2)
a,b,c
```

Output:

$$c + 2, 4, 3$$

Input:

```
eval_level(3)
a,b,c
```

Output:

$$5, 4, 3$$

Input:

```
eval_level()
```

Output:

$$3$$

6.12.3 Evaluating algebraic expressions: evala

In Maple, `evala` is used to evaluate an expression with algebraic extensions. In Xcas, `evala` is not necessary, it behaves like `eval` (see Section 6.12.1 p.203), but it is included for Maple compatibility.

6.12.4 Preventing evaluation: quote hold '

You can prevent an expression from being evaluated by *quoting* it, either by preceding it with `'` or with the `quote` or `hold` command.

Remark.

If a is a variable, then $a := \text{quote}(a)$ (or $a := \text{hold}(a)$) is equivalent to $\text{purge}(a)$ (for the sake of Maple compatibility). It returns the value of this variable (or the hypothesis done on this variable).

Example.

Input:

```
a:=2;quote(2+3*a)
```

or:

```
a:=2;'2+3*a'
```

Output:

$$2, 2 + 3a$$

6.12.5 Forcing evaluation: `unquote`

`unquote` is used for evaluation inside a quoted expression.

For example in an assignment, the variable is automatically quoted (not evaluated) so that the user does not have to quote it explicitly each time he want to modify its value. In some circumstances, you might want to evaluate it.

Input:

```
purge(b);a:=b;unquote(a):=3
```

The variable `b` begins as a purely symbolic variable, and the value of `a` is equal to the symbolic variable `b`. In the assignment `unquote(a):=3`, the left hand side `unquote(a)` is evaluated to `b`, and so `b` is assigned the value 3. Since `a` evaluates to the same thing as `b`, `a` also evaluated to 3.

Input:

```
a,b
```

Output:

```
3,3
```

6.12.6 Distribution: `expand` `fdistrib`

The `expand` command distributes multiplication across addition. `fdistrib` is a synonym for `expand`.

- `expand` takes one argument:
expr, an expression.
- `expand(expr)` returns the expression *expr* with multiplication distributed with respect to addition.

Example.

Input:

```
expand((x+1)*(x-2))
```

or:

```
fdistrib((x+1)*(x-2))
```

Output:

$$x^2 - x - 2$$

6.12.7 Canonical form: `canonical_form`

The canonical form of a second degree polynomial in a variable x is the form $a(x - c)^2 + b$.

The `canonical_form` command finds the canonical form of a second degree polynomial.

- `canonical_form` takes one argument:
p, a second degree polynomial.

- `canonical_form(p)` returns the canonical form of p .

Examples.

- *Input:*

```
canonical_form(x^2-6*x+1)
```

Output:

$$(x - 3)^2 - 8$$

- *Input:*

```
canonical_form(2*t^2+3*t+8)
```

Output:

$$2 \left(t + \frac{3}{4} \right)^2 + \frac{55}{8}$$

6.12.8 Multiplication by the conjugate quantity: `mult_conjugate`

The `mult_conjugate` tries to remove square roots from the bottom of an expression.

- `mult_conjugate` takes one argument:
 $expr$, an expression. The denominator or numerator is supposed to contain a square root.
- `mult_conjugate(expr)` returns the following:
 - If $expr$ is a fraction and the denominator contains a square root, then this expression is returned with the numerator and denominator multiplied by the conjugate of the denominator.
 - If $expr$ is a fraction and the numerator, but not the denominator, contains a square root (if $expr$ is not a fraction, it is regarded as a fraction with a denominator of 1), then this expression is returned with the numerator and denominator multiplied by the conjugate of the numerator.

Examples.

- *Input:*

```
mult_conjugate((2+sqrt(2))/(2+sqrt(3)))
```

Output:

$$\frac{(2 + \sqrt{2})(2 - \sqrt{3})}{(2 + \sqrt{3})(2 - \sqrt{3})}$$

- *Input:*

```
mult_conjugate((2+sqrt(2))/(sqrt(2)+sqrt(3)))
```

Output:

$$\frac{(2 + \sqrt{2})(-\sqrt{2} + \sqrt{3})}{(\sqrt{2} + \sqrt{3})(-\sqrt{2} + \sqrt{3})}$$

- *Input:*

```
mult_conjugate((2+sqrt(2))/2)
```

Output:

$$\frac{(2 + \sqrt{2})(2 - \sqrt{2})}{2(2 - \sqrt{2})}$$

6.12.9 Separation of variables: `split`

The `split` command tries to factor an expression involving two variables into the product of two expressions, each of which depends on only one of the variables.

- `split` takes two arguments:
 - *expr*, an expression depending on two variables *x* and *y*.
 - [*x*, *y*], the list of these two variables.
- `split(expr, [x, y])` returns a list [*factor*₁, *factor*₂], if such a list exists, where *expr*=*factor*₁·*factor*₂, *factor*₁ only depends on *x* and *factor*₂ only depends on *y*. If such a factorization doesn't exist, the list [0] is returned.

Examples.

- *Input:*

```
split((x+1)*(y-2), [x, y])
```

or:

```
split(x*y-2*x+y-2, [x, y])
```

Output:

$$[x + 1, y - 2]$$

- *Input:*

```
split((x^2*y^2-1), [x, y])
```

Output:

$$[0]$$

6.12.10 Factoring: factor cfactor

The **factor** and **cfactor** commands factor expressions over their coefficient fields or extensions of their fields. (See also Section 6.27.16 p.361.)

- **factor** takes one mandatory argument and one optional argument:
 - *expr*, an expression or a list of expressions.
 - Optionally, α , to specify an extension field.
- **factor**(*expr*) returns *expr* factored over the field of its coefficients, with the addition of i in complex mode (see Section 3.5.5 p.72). If **sqr**t is enabled in the Cas configuration (see Section 3.5.7 p.73), polynomials of order 2 are factorized in complex mode or in real mode if the discriminant is positive.
factor(*expr*, α) returns *expr* factored over $F[\alpha]$, where F is the field of coefficients of *expr*.
- **cfactor** factors like **factor**, except the field includes i whether in real or complex mode.

Examples.

- Factor $x^4 - 1$ over \mathbb{Q} .
Input:

```
factor(x^4-1)
```

Output:

$$(x - 1)(x + 1)(x^2 + 1)$$

The coefficients are rationals, hence the factors are polynomials with rational coefficients.

- Factor $x^4 - 1$ over $\mathbb{Q}[i]$.
 This can be done in a number of ways.

- Using **cfactor**.
Input:

```
cfactor(x^4-1
```

- Using **factor** with adding i to the extension field.
Input:

```
factor(x^4-1,i)
```

- Using **factor** in complex mode.
Input (in complex mode):

```
factor(x^4-1)
```

In all cases, the result will be:

Output:

$$(x - 1)(x + 1)(x + i)(x - i)$$

- Factor $x^4 + 1$ over \mathbb{Q}

Input:

```
factor(x^4+1)
```

Output:

$$x^4 + 1$$

Indeed $x^4 + 1$ has no factor with rational coefficients.

- Factor $x^4 + 1$ over $\mathbb{Q}[i]$.

Using complex mode:

Input:

```
cfactor(x^4+1)
```

Output:

$$(x^2 + i)(x^2 - i)$$

- Factor $x^4 + 1$ over \mathbb{R} .

You have to provide the square root required for extending the rationals. In order to do that with the help of **Xcas**, first check **complex** in the **cas** configuration:

Input:

```
solve(x^4+1,x)
```

Output:

$$\left[\frac{1}{2}\sqrt{2}(1-i), -\frac{1}{2}\sqrt{2}(1-i), -\frac{1}{2}\sqrt{2}(1-i)i, \frac{1}{2}\sqrt{2}(1-i)i \right]$$

The roots depend on $\sqrt{2}$, and so will be in $\mathbb{Q}[\sqrt{2}]$. Putting **Xcas** back in real mode, either check the **sqrt** box in the Cas configuration or:

Input:

```
factor(x^4+1,sqrt(2))
```

Output:

$$(x^2 - \sqrt{2}x + 1)(x^2 + \sqrt{2}x + 1)$$

To factor over \mathbb{C} , put **Xcas** back in complex mode or input `cfactor(x^4+1,sqrt(2))`.

6.12.11 Zeros of an expression: zeros

The **zeros** command finds the zeros of an expression.

- **zeros** takes one mandatory argument and one optional argument:
 - *expr*, an expression.

- Optionally, x , a variable name to use (which by default will be x).
- `zeros(expr, x)` returns a list of values of the variable where the expression vanishes. The list may be incomplete in exact mode if the expression is not a polynomial or if intermediate factorizations have irreducible factors of order strictly greater than 2.

In real mode, (which means the complex box is unchecked in the Cas configuration (see Section 3.5.7 p.73) or with `complex_mode:=0`), only reals zeros are returned. With (`complex_mode:=1`), real and complex zeros are returned. `cZeros` behaves like `zeros`, except that it returns complex zeros whether in real or complex mode.

Examples.

- *Input (in real mode):*

`zeros(x^2+4)`

Output:

`[]`

Input (in complex mode):

`zeros(x^2+4)`

Output:

`[-2i, 2i]`

Input (in real or complex mode):

`cZeros(x^2+4)`

Output:

`[-2i, 2i]`

- *Input (in real mode):*

`zeros(ln(x)^2-2)`

Output:

`[e√2, e-√2]`

- *Input (in real mode):*

`zeros(ln(y)^2-2, y)`

Output:

`[e√2, e-√2]`

- *Input (in real mode):*

`zeros(x*(exp(x))^2-2*x-2*(exp(x))^2+4)`

Output:

`[ln(2)/2, 2]`

6.12.12 Regrouping expressions: regroup

The **regroup** command simplifies expressions.

- **regroup** takes one argument:
expr, an expression.
- **regroup**(*expr*) returns *expr* with some straightforward simplifications.

Example.

Input:

```
regroup(x + 3 * x + 5 * 4 / x)
```

Output:

$$4x + \frac{20}{x}$$

6.12.13 Normal form: normal

The **normal** command takes an expression and considers it to be a rational function with respect to generalized identifiers (which are either true identifiers or transcendental functions replaced by temporary identifiers) with coefficients in \mathbb{Q} or $\mathbb{Q}[i]$ or in an algebraic extension (such as $\mathbb{Q}[\sqrt{2}]$ and finds its expanded irreducible representation.

- **normal** takes one argument:
expr, an expression.
- **normal**(*expr*) returns the expanded irreducible representation of *expr*. (See also **ratnormal**, Section 6.12.16 p.215, for pure rational function or **simplify**, Section 6.12.14 p.213, if the transcendental functions are not algebraically independent.)

Examples.

- *Input:*

```
normal((x-1)*(x+1))
```

Output:

$$x^2 - 1$$

- *Input:*

```
normal((1-sin(x))*(1+sin(x)))
```

Output:

$$-\sin^2 x + 1$$

Remarks.

- Unlike **simplify**, **normal** does not try to find algebraic relations between transcendental functions like $\cos(x)^2 + \sin(x)^2 = 1$.
- It is sometimes necessary to run the **normal** command twice to get a fully irreducible representation of an expression containing algebraic extensions.

6.12.14 Simplifying: simplify

The `simplify` command simplifies an expression. It behaves like `normal` for rational functions and algebraic extensions. For expressions containing transcendental functions, `simplify` tries first to rewrite them in terms of algebraically independent transcendental functions. For trigonometric expressions, this requires radian mode (check `radian` in the `cas` configuration, see Section 3.5.7 p.73, or input `angle_radian:=1`).

- `simplify` takes one argument:
expr, an expression.
- `simplify(expr)` returns a simplified version of *expr*.

Examples.

- *Input:*

```
simplify((x-1)*(x+1))
```

Output:

$$x^2 - 1$$

- *Input:*

```
simplify(3-54*sqrt(1/162))
```

Output:

$$-3\sqrt{2} + 3$$

- *Input:*

```
simplify((sin(3*x)+sin(7*x))/sin(5*x))
```

Output:

$$2 \cos(2x)$$

6.12.15 Automatic simplification: autosimplify

The `autosimplify` command determines how much simplification `Xcas` will do automatically when you enter an expression. Note that `autosimplify` only works with `Xcas`, it doesn't work with `icas` or any other frontend.

By default, `Xcas` will apply the `regroup` command (see Section 6.12.12 p.212) to your input, but the `autosimplify` command can change this to applying another rewriting command to your input, such as `simplify` (see Section 6.12.14 p.213), `factor` (see Section 6.12.10 p.209), or even `nop` for no simplification. With no arguments, `autosimplify` will return the current rewriting command. Otherwise:

- `autosimplify` command takes one argument:
cmd, a command that will be used to rewrite the results in `Xcas`.

- `autosimplify(cmd)` will tell Xcas to apply *cmd* to subsequent inputs. To change the simplification mode during a session, the `autosimplify` command should be on its own line.

Examples.

- *Input:*

```
autosimplify(nop)
```

then:

$$1 + x^2 - 2$$

Output:

$$1 + x^2 - 2$$

- *Input:*

```
autosimplify(simplify)
```

then:

$$1 + x^2 - 2$$

Output:

$$x^2 - 1$$

- *Input:*

```
autosimplify(factor)
```

then:

$$1 + x^2 - 2$$

Output:

$$(x - 1)(x + 1)$$

- *Input:*

```
autosimplify(regroup)
```

then:

$$1 + x^2 - 2$$

Output:

$$x^2 - 1$$

6.12.16 Normal form for rational functions: `ratnormal`

The `ratnormal` command rewrites an expression using its irreducible representation. The expression is viewed as a multivariate rational function with coefficients in \mathbb{Q} (or $\mathbb{Q}[i]$). The variables are generalized identifiers which are assumed to be algebraically independent. Unlike with `normal`, an algebraic extension is considered as a generalized identifier. Therefore `ratnormal` is faster but might miss some simplifications if the expression contains radicals or algebraically dependent transcendental functions.

- `ratnormal` takes one argument:
expr, an expression.
- `ratnormal(expr)` returns the irreducible representation of *expr*.

Examples.

- *Input:*

```
ratnormal((x^3-1)/(x^2-1))
```

Output:

$$\frac{x^2 + x + 1}{x + 1}$$

- *Input:*

```
ratnormal((-2x^3+3x^2+5x-6)/(x^2-2x+1))
```

Output:

$$\frac{-2x^2 + x + 6}{x - 1}$$

6.12.17 Substituting a variable by a value: `|`

The `|` operator is an infix operator that evaluates an expression after giving values to some variables. It does not evaluate the expression before the variables are replaced by the requested values.

- `|` is an infix operator, so takes two arguments:
 - *expr*, an expression depending on one or more variables on the left hand side.
 - $x_1 = a_1$, `ldots`; an equality or sequence of several equalities.
- `expr|x1 = a1, ...` returns the expression *expr* with x_1 replaced by a_1 , etc.

Examples.

- *Input:*

```
a^2 + 1 | a = 2
```

Output (even if a has been assigned a value):

5

- *Input:*

$a^2 + b \mid a = 2, b = 3$

Output (even if a or b had been assigned a value):

7

6.12.18 Substituting a variable by a value: `subst`

The `subst` command replaces specified variables in an expression by specified values. Unlike the `|` operator, the `subst` command evaluates the expression before replacing the variables. Since `subst` does not quote its argument, in a normal evaluation process the substitution variable should be purged (see Section 5.4.8 p.104), otherwise it will be replaced by its assigned value before substitution is done.

The `subst` command can specify the values of variables in two different ways.

The first way:

- `subst` takes two arguments.
 - *expr*, an expression.
 - *eqs*, an equation of the form $x = a$, or a list of such equalities.
- `subst(expr, eqs)` returns the expression with the variables replaced by their values.

Examples.

- *Input (if the variable a is purged, otherwise first enter `purge(a)`):*

`subst(a^2+1, a=2)`

Output:

5

- *Input (if the variables a and b are purged, otherwise first enter `purge(a, b)`):*

`subst(a^2+b, [a=2, b=1])`

Output:

5

The second way:

- `subst` takes three arguments.

- *expr*, an expression.
- *vars*, a variable or a list of variables.
- *vals*, a value or a list of values for substitution.
- `subst(expr, vars, vals)` returns the expression with the variables replaced by their values.

Examples.

- *Input (if the variable a is purged, otherwise first enter `purge(a)`):*

`subst(a^2+1, a, 2)`

Output:

5

- *Input (if the variables a and b are purged, otherwise first enter `purge(a, b)`):*

`subst(a^2+b, [a, b], [2, 1])`

Output:

5

`subst` may also be used to make a change of variable in an integral. In this case the `integrate` command (see Section 6.20.1 p.278) should be quoted (see Section 6.12.4 p.205, otherwise, the integral would be computed before substitution) or the inert form `Int` should be used. In both cases, the name of the integration variable must be given as an argument of `Int` or `integrate` even you are integrating with respect to x .

Examples.

- *Input:*

`subst('integrate(sin(x^2)*x,x,0,pi/2)', x=sqrt(t))`

or:

`subst(Int(sin(x^2)*x,x,0,pi/2), x=sqrt(t))`

Output:

$$\int_0^{\frac{\pi^2}{4}} \frac{1}{2} \sin t \cdot \sqrt{t} \sqrt{t}^{-1} dt$$

Input:

`subst('integrate(sin(x^2)*x,x)', x=sqrt(t))`

or:

`subst(Int(sin(x^2)*x,x), x=sqrt(t))`

Output:

$$\int \frac{1}{2} \sin t \cdot \sqrt{t} \sqrt{t}^{-1} dt$$

6.12.19 Substituting a variable by a value: ()

Another way to substitute a variable by a value, besides with the `|` operator or the `subst` command, is with something akin to functional notation. You can follow an expression or expression name with equalities of the form *variable = value*.

Examples.

- *Input:*

$$\text{Expr} := x + 2*y + 3*z$$

then:

$$\text{subst}(\text{Expr}, [x=1, y=2])$$

or:

$$\text{Expr} \mid x=1, y=2$$

or:

$$\text{Expr}(x=1, y=2)$$

Output:

$$5 + 3z$$

- *Input:*

$$(h*k*t^2 + h^3*t^3)(t=2)$$

Output:

$$4hk + 8h^3$$

6.12.20 Substituting a variable by a value (Maple and Mupad compatibility): subs

In **Maple** and in **Mupad**, you would use the `subs` command to substitute a variable by a value in an expression. But the order of the arguments differ between **Maple** and **Mupad**. Therefore, to achieve compatibility, in **Xcas**, the `subs` command arguments order depends on the mode (see Section 3.5.2 p.71).

In **Maple** mode:

- `subs` takes two arguments:
 - *eq*, an equality or list of equalities of the form *var=value*.
 - *expr*, an expression.
- `subs(eq, expr)` returns the expression with the variables replaced by their given values.

Examples.

- *Input in Maple mode (if the variable a is purged, otherwise first enter `purge(a)`):*

$$\text{subs}(a=2, a^2+1)$$

Output:

$$5$$

- *Input in Maple mode (if the variables a and b are purged, otherwise first enter `purge(a, b)`):*

$$\text{subs}([a=2, b=1], a^2+b)$$

Output:

$$5$$

In Mupad or Xcas or TI modes, `subs` behaves like `subst` (see Section 6.12.18 p.216).

- `subst` takes two or three arguments.
 - *expr*, an expression.
 - *eqs*, an equality of the form *var=value* or a list of such equalities, or
vars,vals, a variable or list of variables followed by a value or a list of values for substitution.
- `subst(expr, eqs)` or `subst(expr, vars, vals)` returns the expression with the variables replaced by their given values.

Examples.

- *Input in Mupad or Xcas or TI modes (if the variable a is purged, otherwise first enter `purge(a)`):*

$$\text{subs}(a^2+1, a=2)$$

or:

$$\text{subs}(a^2+1, a, 2)$$

Output:

$$5$$

- *Input in Mupad or Xcas or TI modes (if the variables a and b are purged, otherwise first enter `purge(a, b)` first):*

$$\text{subs}(a^2+b, [a=2, b=1])$$

or:

```
subs(a^2+b,[a,b],[2,1])
```

Output:

5

Note that **subs** does not quote its argument, hence in a normal evaluation process, the substitution variable should be purged otherwise it will be replaced by its assigned value before substitution is done.

6.12.21 Substituting a subexpression by another expression: algsubs

The **algsubs** command replaces subexpressions of an expression, rather than just replace variables.

- **algsubs** takes two arguments:
 - $expr_1=expr_2$, an equation between two expressions.
 - $expr$, another expression.
- **algsubs**($expr_1=expr_2, expr$) returns the last expression $expr$ with **expr**₁ replaced by **expr**₂.

Examples.

- *Input:*

```
algsubs(x^2 = u, 1 + x^2 + x^4)
```

Output:

$$u^2 + u + 1$$

- *Input:*

```
algsubs(a*b/c = d, 2*a*b^2/c)
```

Output:

$$2 * b * d$$

- *Input:*

```
algsubs(2a = p^2-q^2, algsubs (2c = p^2 + q^2, c^2-a^2))
```

Output:

$$p^2q^2$$

6.12.22 Eliminating one or more variables from a list of equations: `eliminate`

The `eliminate` command eliminates variables from a list of equations.

- `eliminate` takes two arguments:
 - *eqns*, a list of equations.
 - *vars*, the variable or list of variables to eliminate. The equations can be given as expressions, in which case they will be assumed to be 0.
- `eliminate(eqns, vars)` returns the equations with the variables *vars* eliminated or an indication that `Xcas` can't eliminate them.

Examples.

Assuming the variables used haven't been set to any values:

- *Input:*

```
eliminate([x = v0*t, y = y0-g*t^2], t)
```

Output:

$$[gx^2 + yv_0^2 - v_0^2y_0]$$

- *Input:*

```
eliminate([x+y+z+t-2, x*y*t=1, x^2+t^2=z^2], [x, z])
```

Output:

$$[2t^2y^2 - 4t^2y + ty^3 - 4ty^2 + 4ty + 2t + 2y - 4]$$

If the variable(s) can't be eliminated, then `eliminate` returns `[1]` or `[-1]`. If `eliminate` returns `[]`, that means the equations determine the values of the variables to be eliminated.

Examples.

- *Input:*

```
x:=2;y:=-5
eliminate([x=2*t, y=1-10*t^2], t)
```

Output:

$$[1]$$

since `t` cannot be eliminated from both equations.

- *Input:*

```
x:=2;y:=-9
eliminate([x=2*t, y=1-10*t^2], t)
```

Output:

\square

since the first equation gives $t = 1$, which satisfies the second equation.

- *Input:*

```
x:= 2; y:= -9
eliminate([x = 2*t, y = 1-10*t^2, z = x + y - t], t)
```

Output:

$[1, z + 8]$

since the first equation gives $t = 1$, which satisfies the second equation, and so that leaves $z = 2 - 9 - 1 = -8$, or $z + 8 = 0$.

6.12.23 Evaluating a primitive at boundaries: `preval`

The `preval` command evaluates an expression from one value to another, such as in done when evaluating a definite integral using the Fundamental Theorem of Calculus.

- `preval` takes three arguments:
 - F , an expression depending on the variable x .
 - a and b , two expressions.
- `preval(F, a, b)` returns $F|_{x=b} - F|_{x=a}$.

`preval` is used to compute a definite integral when the primitive F of the integrand f is known. Assume, for example, that $F := \text{int}(f, x)$, then `preval(F, a, b)` is equivalent to `int(f, x, a, b)`, but does not require you to recompute F from f if you change the values of a or b .

Example.

Input:

```
preval(x^2+x,2,3)
```

Output:

6

6.12.24 Sub-expression of an expression: `part`

The `part` command finds subexpressions of an expression. (See Section 4.3.2 p.89.)

- `part` takes two arguments:
 - $expr$, an expression.
 - n , an integer.
- `part($expr, n$)` evaluates $expr$ and then returns the n th sub-expression of $expr$.

Examples.

- *Input:*

$$\text{part}(x^2+x+1, 2)$$

Output:

$$x$$

- *Input:*

$$\text{part}(x^2+(x+1)*(y-2)+2, 2)$$

Output:

$$(x+1)(y-2)$$

- *Input:*

$$\text{part}((x+1)*(y-2)/2, 2)$$

Output:

$$y-2$$

6.13 Values of a sequence u_n

6.13.1 Array of values of a sequence : `tablefunc`

The `tablefunc` command fills two columns of a spreadsheet with a table of values of a function. The spreadsheet can be opened with `Alt+t` (see Section 4.5 p.92).

- `tablefunc` takes four arguments:

- $f(x)$, a formula for a function.
- x , the variable.
- x_0 , the beginning value of x .
- inc , an increment for x .

- `tablefunc(f(x), x, x0, inc)` fills two columns of the spreadsheet, the current column and the following column, starting with the chosen cell. The current column starts with the variable x , followed by the initial value x_0 , then x_0+inc , x_0+2inc , ... The following column starts with the formula $f(x)$, followed by $f(x)$ evaluated at the values in the first column. (If the current cell is column C , row n , it will contain x , the cell below it will contain inc , and the cell below it in row k will contain $=C(k-1) + C$(n+1)$, and the corresponding cells in the next column will contain $=evalf(subst(D$n, C$n, Ck)).$)

Example.

Display the values of the sequence $u_n = \sin(n)$

Select a cell of a spreadsheet (for example C0) and:

Input:

```
tablefunc(sin(n),n,0,1)
```

Output:

row	C	D
	n	sin(n)
	0	0.0
	1	0.841470984808
	2	0.909297426826
	3	0.14112000806
	4	-0.756802495308
	\vdots	\vdots

The graphic representation may be plotted with the `plotfunc` command (see 8.4.1).

6.13.2 Values of a recurrence relation or a system: `seqsolve`

(See also Section 6.13.3 p.225.)

The `seqsolve` command finds the terms of a recurrence relation.

- `seqsolve` takes three arguments:
 - *exprs*, an expression or list of expressions that define the recurrence relation.
 - *vars*, a list of the variables used.
 - *a*, the starting value.
- `seqsolve(exprs,vars,a)` returns a formula for the n th term of the sequence.

For example, if a recurrence relation is defined by $u_{n+1} = f(u_n, n)$ with $u_0 = a$, the arguments to `seqsolve` will be $f(x, n)$, $[x, n]$ and a . If the recurrence relation is defined by $u_{n+2} = g(u_n, u_{n+1}, n)$ with $u_0 = a$ and $u_1 = b$, the arguments to `seqsolve` will be $g(x, y, n)$, $[x, y, n]$ and $[a, b]$.

The recurrence relation must have a homogeneous linear part, the non-homogeneous part must be a linear combination of a polynomials in n times geometric terms in n .

Examples.

- Find u_n , given that $u_{n+1} = 2u_n + n$ and $u_0 = 3$.
Input:

```
seqsolve(2x+n,[x,n],3)
```


Output:

$$-n - 1 + 4 \cdot 2^n$$

- Find u_n , given that $u_{n+1} = 2u_n + n3^n$ and $u_0 = 3$.

Input:

$$\text{seqsolve}(2x+n*3^n, [x, n], 3)$$

Output:

$$(n - 3) \cdot 3^n + 6 \cdot 2^n$$

- Find u_n , given that $u_{n+1} = u_n + u_{n-1}$, $u_0 = 0$ and $u_1 = 1$.

Input:

$$\text{seqsolve}(x+y, [x, y, n], [0, 1])$$

Output:

$$\frac{5 \left(\frac{-\sqrt{5}+1}{2} \right)^n - 4 \left(\frac{-\sqrt{5}+1}{2} \right)^n \sqrt{5} - 5 \left(\frac{-\sqrt{5}+1}{2} \right)^n + 5 \left(\frac{\sqrt{5}+1}{2} \right)^n + 4 \left(\frac{\sqrt{5}+1}{2} \right)^n \sqrt{5} - 5 \left(\frac{\sqrt{5}+1}{2} \right)^n}{20}$$

- Find u_n and v_n , given that $u_{n+1} = u_n + 2v_n$, $v_{n+1} = u_n + n + 1$ with $u_0 = 1, v_0 = 1$.

Input:

$$\text{seqsolve}([x+2*y, n+1+x], [x, y, n], [0, 1])$$

Output:

$$\left[\frac{-2n - (-1)^n + 4 \cdot 2^n - 3}{2}, \frac{(-1)^n + 2 \cdot 2^n - 1}{2} \right]$$

6.13.3 Values of a recurrence relation or a system: **rsolve**

(See also Section 6.13.2 p.224.)

The **rsolve** command is an alternate way to find the values of a recurrence relation. Note that **rsolve** is more flexible than **seqsolve** since:

- The sequence doesn't have to start with u_0 .
- The sequence can have several starting values, such as initial condition $u_0^2 = 1$, which is why **rsolve** returns a list.
- The notation for the recurrence relation is similar to how it is written in mathematics.
- **rsolve** takes three arguments:
 - *eqns*, an equation or list of equations that define the recurrence relation.

- *fns*, the function or list of functions (with their variables) used.
- *startvals*, the equation or list of equations for the starting values.
- **rsolve**(*eqns*, *fns*, *startvals*) returns a list containing a formula for the n th term of the sequence. (If there is more than one sequence, it will return a formula for each one.)

For example, if a recurrence relation is defined by $u_{n+1} = f(u_n, n)$ with $u_0 = a$, the arguments to **rsolve** will be $u(n+1) = f(u(n), n)$, $u(n)$ and $u(0) = a$.

The recurrence relation must either be a homogeneous linear part with a nonhomogeneous part being a linear combination of polynomials in n times geometric terms in n (such as $u_{n+1} = 2u_n + n3^n$), or a linear fractional transformation (such as $u_{n+1} = (u_n - 1)/(u_n - 2)$).

Examples.

- Find u_n , given that $u_{n+1} = 2u_n + n$ and $u_0 = 3$.

Input:

```
rsolve(u(n+1) = 2*u(n) + n, u(n), u(0)=3)
```

Output:

$$[-n + 4 \cdot 2^n - 1]$$

- Find u_n , given that $u_{n+1} = 2u_n + n$ and $u_1^2 = 1$.

Input:

```
rsolve(u(n+1) = 2*u(n) + n, u(n), u(1)^2 = 1)
```

Output:

$$\left[-n + \frac{3}{2} \cdot 2^n - 1, -n + \frac{1}{2} \cdot 2^n - 1 \right]$$

Note that there are two formulas, since the starting formula $u_1^2 = 1$ gives two possible starting values: $u_1 = 1$ and $u_1 = 2$.

- Find u_n , given that $u_{n+1} = 2u_n + n3^n$ and $u_0 = 3$.

Input:

```
rsolve(u(n+1) = 2*u(n) + n*3^n, u(n), u(0)=3)
```

Output:

$$[n \cdot 3^n + 6 \cdot 2^n - 3 \cdot 3^n]$$

- Find u_n , given that $u_{n+1} = (u_n - 1)/(u_n - 2)$ and $u_0 = 4$.

Input:

```
rsolve(u(n+1) = (u(n)-1)/(u(n)-2), u(n), u(0)=4)
```

Output:

$$\left[\frac{(20\sqrt{5} + 60) \left(\frac{\sqrt{5}-3}{2}\right)^n + 60\sqrt{5} - 140}{40 \left(\frac{\sqrt{5}-3}{2}\right)^n + 20\sqrt{5} - 60} \right]$$

- Find u_n given that $u_{n+1} = u_n + u_{n-1}$ with $u_0 = 0, u_1 = 1$.

Input:

`rsolve(u(n+1) = u(n) + u(n-1), u(n), u(0) = 0, u(1) = 1)`

Output:

$$\left[-\frac{\sqrt{5}}{5} \left(\frac{-\sqrt{5}+1}{2}\right)^n + \frac{1}{5}\sqrt{5} \left(\frac{\sqrt{5}+1}{2}\right)^n \right]$$

- To find u_n and v_n , given that $u_{n+1} = u_n + v_n, v_{n+1} = u_n - v_n$ with $u_0 = 0, v_0 = 1$.

Input:

`rsolve([u(n+1) = u(n) + v(n), v(n+1) = u(n) - v(n)],
[u(n), v(n)], [u(0)=1, v(0)=1])`

Output:

$$\left[\frac{1}{2} (-\sqrt{2} + 1) (-\sqrt{2})^{n+1-1} + \frac{1}{2} (\sqrt{2} + 1) \cdot 2^{\frac{n+1-1}{2}} \quad \frac{1}{2} (-\sqrt{2})^{n+1-1} + \frac{1}{2} \cdot 2^{\frac{n+1-1}{2}} \right]$$

6.13.4 Table of values and graph of a recurrent sequence: `tableseq`

The `tableseq` command fills a column of a spreadsheet with a recurrence relation. The spreadsheet can be opened with `Alt+t` (see Section 4.5 p.92).

`tableseq` takes three arguments, which can be different depending on how many terms are involved in the recurrence relation.

For a one term recurrence relation:

- `tableseq` takes three arguments:

- $f(x)$, a formula which defines the recurrence, through $u_{n+1} = f(u_n)$.
- x , the variable.
- u_0 , the initial term of the sequence.

- `tableseq(f(x), x, u_0)` fills the current column of the spreadsheet, starting with the selected cell (or cell 0 if the entire column is selected), with the formula $f(x)$, the next cell with the variable x , followed by the terms u_0, u_1, \dots of the sequence. (If the current cell is column C , row n , these latter cells will actually contain (if in row k) `=evalf(subst(Cn, C(n+1), C$(k-1)))`, which means if you change the value in one cell, the values in the later cells will change accordingly.) See also `plotseq`, Section 8.17 p.697, for a graphic representation of a one-term recurrence sequence.

Example. Display the values of the sequence $u_0 = 3.5, u_{n+1} = \sin(u_n)$. Select a cell of the spreadsheet (for example B0) and input in the command line:

`tableseq(sin(x),x,3.5)`

Output:

row	B
0	sin(x)
1	x
2	3.5
3	-0.35078322769
4	-0.343633444925
5	-0.336910330426
...	...

More generally, for a recurrence relation where each term depends on the previous k terms:

- `tableseq` takes three arguments:
 - $f(x_1, x_2, \dots, x_k)$, a formula which defines the recurrence, through $u_{n+1} = f(u_n, \dots, u_{n-k})$.
 - $[x_1, \dots, x_k]$, a list of variables.
 - $[u_0, \dots, u_{k-1}]$, a list of the beginning k terms.
- `tableseq($f(x_1, \dots, x_k)$, $[x_1, \dots, x_k]$, $[u_0, \dots, u_{k-1}]$)` fills the current column of the spreadsheet, starting with the selected cell (or cell 0 if the entire column is selected), with the formula $f(x_1, x_2, \dots, x_k)$, followed by the variables x_1, \dots, x_k , followed by the terms u_0, u_1, \dots of the sequence.

Example.

Display the values of the Fibonacci sequence $u_0 = 1, u_1 = 1, \dots, u_{n+2} = u_n + u_{n+1}$

Select a cell, say B0, and:

Input:

`tableseq(x+y, [x,y], [1,1])`

Output:

row	B
0	x+y
1	x
2	y
3	1
4	1
5	2
...	...

6.14 Operators or infix functions

An operator is an infix function. For example, the arithmetic functions $+$, $-$, $*$, $/$, and $^$ are operators. (See Section 6.8.2 p.173 and Section 6.10.1 p.197.)

6.14.1 Xcas operators: \$ %

- $\$$ is the infix version of `seq` (see Section 6.39.2 p.454).

Example.

Input:

$$(2^k)\$(k=0..3)$$

(do not forget to put parenthesis around the arguments)

or:

$$\text{seq}(2^k, k=0..3)$$

Output:

$$1, 2, 4, 8$$

- `mod` or `%` defines a modular number; $a \bmod n$ is the equivalence class of a in $\mathbb{Z}/n\mathbb{Z}$.

Example.

Input:

$$5 \% 7$$

or:

$$5 \bmod 7$$

Output:

$$(-2) \% 7$$

- `@` is used to compose functions; $(f@g)(x) = f(g(x))$.

Example.

Input:

$$(\sin @ \exp)(x)$$

Output:

$$\sin(e^x)$$

- `@@` is used to compose a function with itself many times (like a power, replacing multiplication by composition); for example, $(f@@3)(x) = f(f(f(x)))$.

Example.

Input:

```
(sin@@4)(x)
```

Output:

```
sin(sin(sin(sin x)))
```

- `minus`, `union` and `intersect` return the difference, the union and the intersection of two sets, respectively. (See Section 5.3.2 p.98).

Example.

Input:

```
A := set[1,2,3,4];
B := set[3,4,5,6];
```

then:

```
A minus B
```

Output:

```
[[1,2]]
```

Input:

```
A union B
```

Output:

```
[[1,2,3,4,5,6]]
```

then:

```
A intersect B
```

Output:

```
[[3,4]]
```

- `->` is used to define a function, which can be assigned a name.

Example.

Input:

```
(x->x^2)(3)
```

Output:

```
9
```

Input:

```
f := x -> x^2
```

then:

```
f(3)
```

Output:

9

- `=>` is the infix version of `sto` (see Section 5.4.2 p.100) and so is used to store an expression in a variable.

Example.

Input:

`2 => a`

then:

a

Output:

2

- `:=` is used to store an expression in a variable, but the variable comes first (the argument order is switched from `=>`).

Example.

Input:

`a := 2`

then:

a

Output:

2

- `=<` to store an expression in a variable, but the storage is done by reference if the target is a matrix element or a list element. This is faster if you modify objects inside an existing list or matrix of large size, because no copy is made, the change is done in place. Use with care, all objects pointing to this matrix or list will be modified.

Example.

Input:

`L := [2,3];`
`L2 := L;`

then:

`L[0] =< 5`

and:

L

Output:

[5, 3]

Input:

L2

Output:

[5, 3]

6.14.2 Defining an operator: `user_operator`

The `user_operator` command lets you define an operator or delete an operator you previously defined. When you use an operator you defined, you have to make sure that you leave spaces around the operator.

To define an operator:

- `user_operator` takes three arguments:
 - *name*, a string which is the name of the operator.
 - *fn*, a function of one or two variables with values in \mathbb{R} or in `true`, `false`.
 - *type*, to specify what kind of an operator you are defining. The possible values are:
 - * `Binary`, to define an infix operator. In this case, *fn* must be a function of two variables.
 - * `Prefix` (or `Unary`), to define a prefixed operator. In this case, *fn* must be a function of one variable.
 - * `Postfix`, to define a postfix operator. In this case *fn* must be a function of one variable.
- `user_operator(name,fn,type)` returns 1 if the definition was successful and otherwise returns 0.

Examples.

- **Example 1.**

Let R be defined on $\mathbb{R} \times \mathbb{R}$ by $x R y = x * y + x + y$.

To define R :

Input:

```
user_operator("R", (x,y)->x*y+x+y,Binary)
```

Output:

1

Input:

5 R 7

(Do not forget to put spaces around \mathbb{R} .)

Output:

47

• **Example 2.**

Let S be defined on \mathbb{N} by:

for x and y integers, $x S y$ means that x and y are *not* coprime.

To define S :

Input:

```
user_operator("S", (x,y)->(gcd(x,y))!=1,Binary)
```

Output:

1

Input:

5 S 7

(Do not forget to put spaces around \mathbb{S} .)

Output:

false

Input:

8 S 12

Do not forget to put spaces around \mathbb{S} .

Output:

true

• **Example 3.**

Let T be defined on \mathbb{R} by $Tx = x^2$.

To define T :

Input:

```
user_operator("T", x->x^2,Prefix)
```

Output:

1

Input:

T 4

(Do not forget to put a space before \mathbb{T} .)

Output:

16

- **Example 4.**

Let U be defined on \mathbb{R} by $xU = 5x$.

To define U :

Input:

```
user_operator("U",x->5*x,Postfix)
```

Output:

1

Input:

3 U

(Do not forget to put a space before T.)

Output:

15

To delete an operator:

- `user_operator` takes two arguments:

- *name*, a string which is the name of the operator.
- `Delete`

- `user_operator(name,Delete)` deletes the operator.

6.15 Functions and expressions with symbolic variables

6.15.1 The difference between a function and an expression

Functions are often defined with expressions; for example, $f(x) := x^2 - 1$ defines a function f , whose value at x is given by $x^2 + 1$. (The function f can also be defined by $f := x \rightarrow x^2 - 1$.) But the function is not the same as the expression; the variable x is only a placeholder for the function; it is not part of actual definition of the function. Compare this with $g := x^2 - 1$, where g is a variable which stores the expression $x^2 - 1$ and so the identifier x is part of the definition of g . To find the value of f for $x = 2$, you can enter $f(2)$, but to use g to find the same value you have to do an explicit substitution and enter $\text{subst}(g, x=2)$.

When a command expects a function as argument, this argument should be either the definition of the function (e.g. $x \rightarrow x^2 - 1$) or a variable name assigned to a function (e.g. f previously defined by $f(x) := x^2 - 1$).

When a command expects an expression as argument, this argument should be either the definition of the expression (for example $x^2 - 1$), or a variable name assigned to an expression (e.g. g previously defined by $g := x^2 - 1$), or the evaluation of a function (e.g. $f(x)$ where f is the previously defined function by $f(x) := x^2 - 1$).

6.15.2 Transforming an expression into a function: unapply

The `unapply` command transforms an expression into a function.

- `unapply` takes two arguments:
 - $expr$, an expression.
 - x , the name of a variable or sequence of names of variables.
- `unapply($expr, x$)` returns the function defined by the expression $expr$ and variable(s) x , as in $x \mapsto expr$.

Examples.

- *Input:*

```
unapply(exp(x+2), x)
```

Output:

$$x \mapsto e^{x+2}$$

- *Input:*

```
unapply(x*y-x-y, (x, y))
```

Output:

$$(x, y) \mapsto xy - x - y$$

Warning.

When a function being is defined, the right side of the assignment is not evaluated, hence `g:=sin(x+1); f(x):=g` does not define the function $f : x \rightarrow \sin(x+1)$ but defines the function $f : x \rightarrow g$. To define the former function, `unapply` should be used, as in the following example:

Example.

Input:

```
g:= sin(x+1); f:=unapply(g,x)
```

Output:

$$\sin(x+1), x \mapsto \sin(x+1)$$

hence, the variable `g` is assigned to a symbolic expression and the variable `f` is assigned to a function.

Examples.

- *Input:*

```
f:=unapply(lagrange([1,2,3],[4,8,12]), x)
```

(See Section 6.27.29 p.369.) *Output:*

$$x \mapsto 4(x-1) + 4$$

- *Input:*

```
f:=unapply(integrate(log(t),t,1,x),x)
```

Output:

$$x \mapsto x \ln x - x + 1$$

- *Input:*

```
f:=unapply(integrate(log(t),t,1,x),x);  
f(x)
```

Output:

$$x \ln x - x + 1$$

Remark.

Suppose that f is a function of 2 variables $f : (x, w) \rightarrow f(x, w)$, and that g is the function defined by $g : w \rightarrow h_w$, where h_w is the function defined by $h_w(x) = f(x, w)$.

`unapply` can also be used to define g .

Example.

Input:

```
f(x,w):=2*x+w;  
g(w):=unapply(f(x,w),x);  
g(3)
```

Output:

$$x \mapsto 2x + 3$$

6.15.3 Top and leaves of an expression: `sommet` `feuille` `op` `left` `right`

An expression can be represented by a tree. The top of the tree is either an operator or a function and the leaves of the tree are the arguments of the operator or function (see also [6.39.10](#)).

The `sommet` command finds the top of an expression.

- `sommet` takes one argument:
 $expr$, an expression.
- `sommet(expr)` returns the top of $expr$.

Examples.

- *Input:*

```
sommet(sin(x+2))
```

Output:

`sin`

- *Input:*

`sommet(x+2*y)`

Output:

$+$

The `op` command finds the list of the leaves of an expression. *feuille* is a synonym for `op`.

- `op` takes one argument:
expr, an expression.
- `op(expr)` returns the leaves of *expr*.

Examples.

- *Input:*

`op(sin(x+2))`

or:

`feuille(sin(x+2))`

Output:

$x + 2$

- *Input:*

`op(x+2*y)`

or:

`feuille(x+2*y)`

Output:

$x, 2y$

If the top of an expression *expr* is an infix operator, the left hand side will be *expr*[1] and the right hand side will be *expr*[2]. The `left` and `right` commands are alternative commands to find the sides (see Section 6.3.4 p.123, Section 6.37.1 p.445, Section 6.38.2 p.448, Section 6.40.6 p.467, Section 6.55.4 p.618 and Section 6.55.5 p.618 for specific uses of `left` and `right`.)

- `left` and `right` take one argument:
expr, an expression whose top is an infix operator.
- `left(expr)` returns the left side of the operator.

- `right(expr)` returns the right side of the operator.

Examples.

- *Input:*

$$\text{sommet}(y=x^2)$$
Output:

$$=$$

- *Input:*

$$\text{left}(y=x^2)$$
Output:

$$y$$

- *Input:*

$$\text{right}(y=x^2)$$
Output:

$$x^2$$
Remark.

If a function is defined by a program (see Section 12.1.2 p.835) then the top will be the function '`program`' and the leaves will be a sequence consisting of the arguments of the defined function, followed by a sequence of 0s (one for each argument) followed by the body of the function. For example, define the `pgcd` function:

```
pgcd(a,b):={local r; while (b!=0) {r:=irem(a,b);a:=b;b:=r;}  
            return a;}
```

Then:

Input:

$$\text{sommet}(\text{pgcd})$$
Output:

$$\text{program}$$
Input:

$$\text{feuille}(\text{pgcd})[0]$$
or:

$$\text{op}(\text{pgcd})[0]$$
Output:

$$a, b$$
Input:

```
feuille(pgcd)[1]
```

or:

```
op(pgcd)[1]
```

Output:

```
0,0
```

Input:

```
feuille(pgcd)[2]
```

or:

```
op(pgcd)[2]
```

Output:

```
{ local r;
  while(b<>0){
    r:=irem(a,b);
    a:=b;
    b:=r;
  }; ;
  return(a);
}
```

6.16 Functions

6.16.1 Context-dependent functions.

The + operator

The + operator is infix and '+' is its prefixed version. The + operator will add numbers (see Section 6.8.2 p.173), concatenate strings (see Section 6.3.12 p.128), and convert a number to a string if necessary. Addition makes sense for other objects, and + can flexibly deal with them; the result of using the + operator depends on the nature of its arguments.

Examples.

- *Input:*

```
1+2+3+4
```

or:

```
'+'(1,2,3,4)
```

or:

```
(1,2)+(3,4)
```

or:

$$(1, 2, 3) + 4)$$

Output:

$$10$$

(See Section 6.39.9 p.462.)

• *Input:*

$$1 + i + 2 + 3 * i$$

or:

$$'+'(1, i, 2, 3 * i)$$

Output:

$$3 + 4i$$

• *Input:*

$$[1, 2, 3] + [4, 1]$$

or:

$$[1, 2, 3] + [4, 1, 0]$$

or:

$$'+'([1, 2, 3], [4, 1])$$

Output:

$$[5, 3, 3]$$

• *Input:*

$$[1, 2] + [3, 4]$$

or:

$$'+'([1, 2], [3, 4])$$

Output:

$$[4, 6]$$

• *Input:*

$$[[1, 2], [3, 4]] + [[1, 2], [3, 4]]$$

Output:

$$\begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix}$$

- *Input:*

$$[1, 2, 3] + 4$$

or:

$$'+'([1, 2, 3], 4)$$

Output:

$$\llbracket 1, 2, 7 \rrbracket$$

(This is a polynomial; see Section 6.27.1 p.351.)

- *Input:*

$$[1, 2, 3] + (4, 1)$$

or:

$$'+'([1, 2, 3], 4, 1)$$

Output:

$$\llbracket 1, 2, 8 \rrbracket$$

- *Input:*

$$\text{"Hel"} + \text{"lo"}$$

or:

$$'+'(\text{"Hel"}, \text{"lo"})$$

Output:

$$\text{"Hello"}$$

The -, * and / operators

The -, * and / operators (and their prefixed versions '-', '* and '/'), like the + operator, are flexible and operate on compound objects (such as lists and sequences), but don't concatenate strings.

Examples of - and '-'.

- *Input:*

$$(1, 2) - (3, 4)$$

Output:

$$-4$$

- *Input:*

$$(1, 2, 3) - 4$$

Output:

$$2$$

• *Input:*

$$[1, 2, 3] - [4, 1]$$

or:

$$[1, 2, 3] - [4, 1, 0]$$

or:

$$\text{'-'}([1, 2, 3], [4, 1])$$

Output:

$$[-3, 1, 3]$$

• *Input:*

$$[1, 2] - [3, 4]$$

or:

$$\text{'-'}([1, 2], [3, 4])$$

Output:

$$[-2, -2]$$

• *Input:*

$$[[3, 4], [1, 2]] - [[1, 2], [3, 4]]$$

Output:

$$\begin{bmatrix} 2 & 2 \\ -2 & -2 \end{bmatrix}$$

• *Input:*

$$[1, 2, 3] - 4$$

or:

$$\text{'-'}([1, 2, 3], 4)$$

Output:

$$\llbracket 1, 2, -1 \rrbracket$$

• *Input:*

$$[1, 2, 3] - (4, 1)$$

Output:

$$\llbracket 1, 2, -2 \rrbracket$$

Examples of $*$ and $'*'$.

• *Input:*

$$(1, 2) * (3, 4)$$

or:

$$(1, 2, 3) * 4$$

or:

$$1 * 2 * 3 * 4$$

or:

$$'*' (1, 2, 3, 4)$$

Output:

$$24$$

• *Input:*

$$1 * i * 2 * 3 * i$$

or:

$$'*' (1, i, 2, 3 * i)$$

Output:

$$-6$$

• *Input:*

$$[10, 2, 3] * [4, 1]$$

or:

$$[10, 2, 3] * [4, 1, 0]$$

or:

$$'*' ([10, 2, 3], [4, 1])$$

Output:

42

These compute the scalar product.

• *Input:*

$[1, 2] * [3, 4]$

or:

$'*'([1, 2], [3, 4])$

Output:

11

These compute the scalar product.

• *Input:*

$[[1, 2], [3, 4]] * [[1, 2], [3, 4]]$

Output:

$\begin{bmatrix} 7 & 10 \\ 15 & 22 \end{bmatrix}$

• *Input:*

$[1, 2, 3] * 4$

or:

$'*'([1, 2, 3], 4)$

Output:

$[4, 8, 12]$

• *Input:*

$[1, 2, 3] * (4, 2)$

or:

$'*'([1, 2, 3], 4, 2)$

or:

$[1, 2, 3] * 8$

Output:

$[8, 16, 24]$

- *Input:*

$$(1,2)+i*(2,3)$$

or:

$$1+2+i*2*3$$

Output:

$$3 + 6i$$

Examples of / and ' /'.

- *Input:*

$$[10,2,3]/[4,1]$$

Output:

$$\left[\frac{5}{2}, 2 \right]$$

- *Input:*

$$[1,2]/[3,4]$$

or:

$$'/'([1,2],[3,4])$$

Output:

$$\left[\frac{1}{3}, \frac{1}{2} \right]$$

- *Input:*

$$1/[[1,2],[3,4]]$$

or:

$$'/'(1,[[1,2],[3,4]])$$

Output:

$$\begin{bmatrix} -2 & 1 \\ \frac{3}{2} & -\frac{1}{2} \end{bmatrix}$$

- *Input:*

$$[[1,2],[3,4]]*1/ [[1,2],[3,4]]$$

Output:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

- *Input:*

$$[[1, 2], [3, 4]] / [[1, 2], [3, 4]]$$

Output:

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

(This is term-by-term division.)

6.16.2 Standard functions

- The **max** command finds the maximum of a sequence of real numbers.
 - **max** takes an arbitrary number of arguments:
seq, a sequence (or list) of real numbers.
 - **max(seq)** returns the largest number in the sequence *seq*.

Example.

Input:

$$\text{max}(0, 1, 2, -1, -2)$$

Output:

$$2$$

- The **min** command finds the minimum of a sequence of real numbers.
 - **min** takes an arbitrary number of arguments:
seq, a sequence (or list) of real numbers.
 - **min(seq)** returns the smallest number in the sequence *seq*.

Example.

Input:

$$\text{min}(0, 1, 2, -1, -2)$$

Output:

$$-2$$

- The **abs** command finds the absolute value of a real or complex number.
 - **abs** takes one argument:
x, a real or complex number.
 - **abs(x)** returns the absolute value of *x*.

Examples.

– *Input:*

`abs(-5)`

Output:

5

– *Input:*

`abs(3+4*i)`

Output:

5

- The `sign` command finds the sign of a real number (+1 if it is positive, 0 if it is zero, and -1 if it is negative).

– `sign` takes one argument:

x , a real number.

– `sign(x)` returns the sign of x .

Examples.

– *Input:*

`sign(-4)`

Output:

-1

– *Input:*

`sign(0)`

Output:

0

- The `floor` command finds the floor of a real number; namely, the largest integer less than or equal to the number. `iPart`) is a synonym for `floor`.

– `floor` takes one argument:

x , a real number.

– `floor(x)` returns the floor of x .

Examples.

– *Input:*

`floor(4.1)`

Output:

4

– *Input:*

`floor(-4.1)`

Output:

−5

- The **round** command rounds a number to the nearest integer, rounding up in the case of a half-integer.
 - **round** takes one argument:
 x , a real number.
 - **round**(x) returns the nearest integer to x .

Examples.

– *Input:*

round(3.4)

Output:

3

– *Input:*

round(-3.4)

Output:

−3

– *Input:*

round(3.5)

Output:

4

- The **ceil** command finds the ceiling of a real number; namely, the smallest integer greater than or equal to the number.
ceiling is a synonym for **ceil**.
 - **ceil** takes one argument:
 x , a real number.
 - **ceil**(x) returns the ceiling of x .

Examples.

– *Input:*

ceiling(4.1)

Output:

5

– *Input:*

ceiling(-4.1)

Output:

−4

- The **frac** command finds the fractional part of a number; informally, the part of the number to the right of the decimal point with the appropriate plus or minus sign. For a positive real number x , the fractional part is x minus the floor of x ; for a negative real number x , the fractional part is x minus the ceiling of x .
fPart is a synonym for **frac**.

- **frac** takes one argument:
 x , a real number.
- **frac**(x) returns the fractional part of x .

Examples.

- *Input:*

frac(3.24)

Output:

0.24

- *Input:*

frac(-3.24)

Output:

-0.24

- The **trunc** command truncates a real number; namely, it removes the fractional part. The truncated number added to the fractional part will equal the original number.
 - **trunc** takes one argument:
 x , a real number.
 - **trunc**(x) returns the truncated value of x .

Examples.

- *Input:*

trunc(3.24)

Output:

3

- *Input:*

trunc(-3.24)

Output:

-3

- The **id** command is the identity function.
 - **id** takes one argument or a sequence of arguments:
 seq , whose elements can be any type.

- `id(seq)` returns `seq`.

Example.

- *Input:*

$$\text{id}(a, 1, \text{"abc"}, [1, 2, 3])$$
Output:

$$a, 1, \text{"abc"}, [1, 2, 3]$$

- The `sq` command squares its argument.
 - `sq` takes one argument:
 x , any object that can be multiplied by itself.
 - `sq(x)` returns x^2 .

Examples.

- *Input:*

$$\text{sq}(5)$$
Output:

$$25$$

- *Input:*

$$\text{sq}(x+y)$$
Output:

$$(x + y)^2$$

- *Input:*

$$\text{sq}([[1, 2], [3, 4]])$$

(This is a matrix product; see Section 6.44 p.502).

Output:

$$\begin{bmatrix} 7 & 10 \\ 15 & 22 \end{bmatrix}$$

- *Input:*

$$\text{sq}([1, 2, 3])$$
(This is the dot product of $[1, 2, 3]$ with itself.)*Output:*

$$14$$

- The `sqrt` command finds the square root of its argument.
 - `sqrt` takes one argument:
 x , any object for which the $1/2$ power makes sense.
 - `sqrt(x)` returns $x^{1/2}$.

Examples.

– *Input:*

$$\text{sqrt}(9)$$

Output:

$$3$$

– *Input:*

$$\text{sqrt}((x+y)^2)$$

Output:

$$|x + y|$$

– *Input:*

$$\text{simplify}(\text{sqrt}([2,3],[3,5]))$$

Output:

$$\begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix}$$

- The **surd** command finds roots of quantities.
 - **surd** takes two arguments:
 x and n , numbers.
 - **surd**(x,n) returns the n th root of x ; i.e., $x^{1/n}$.

Example.

- *Input:*

$$\text{surd}(15.625,3)$$

Output:

$$2.5$$

- The **exp** command computes the exponential function.
 - **exp** takes one argument:
 x , a number.
 - **exp**(x) returns e^x .

Example.

Input:

$$\text{exp}(1.0)$$

Output:

$$2.71828182846$$

- The `log` command computes the natural logarithm function.
`ln` is a synonym for `log`.
 - `log` takes one argument:
 x , a number.
 - `log(x)` returns the natural logarithm of x .

Example.*Input:*`log(2.0)`*Output:*

0.69314718056

- The `log10` computes the the base-10 logarithm.
 - `log10` takes one argument:
 x , a number.
 - `log10(x)` returns the base-10 logarithm of x .

Example.*Input:*`log10(1000)`*Output:*

3

- The `logb` computes the logarithm to a specified base.
 - `logb` takes two arguments:
 x and b , non-zero numbers.
 - `logb(x , b)` returns the base- b logarithm of x .

Example.*Input:*`logb(10.0,2)`*Output:*

3.32192809489

- The standard trigonometric functions:
 - The `sin` command is the sine function.
 - The `cos` command is the cosine function.
 - The `tan` command is the tangent function (`tan(x)= sin(x)/cos(x)`).
 - The `cot` command is the cotangent function (`cot(x)= cos(x)/sin(x)`).

- The `sec` command is the secant function (`sec(x) = 1/cos(x)`).
- The `csc` command is the cosecant function (`csc(x) = 1/sin(x)`).
- These commands take one argument: x , a number.
The number x will by default represent an angle measured in radians, but you can set `Xcas` to use degrees (see Section 3.5.3 p.71) by setting the variable `angle_radian` to 0; resetting it to 1 will change the angle measure to radians again.
- `sin(x)` returns the sine of x .

Examples.

* *Input (with `angle_radian` equal to 1):*

`sin(pi/4)`

Output:

$$\frac{\sqrt{2}}{2}$$

* *Input (with `angle_radian` equal to 0):*

`sin(30)`

Output:

$$\frac{1}{2}$$

- `cos(x)` returns the cosine of x .

Examples.

* *Input (with `angle_radian` equal to 1):*

`cos(pi/6)`

Output:

$$\frac{\sqrt{3}}{2}$$

* *Input (with `angle_radian` equal to 0):*

`cos(90)`

Output:

$$0$$

- `tan(x)` returns the tangent of x .

Examples.

* *Input (with `angle_radian` equal to 1):*

`tan(pi/4)`

Output:

$$1$$

* *Input (with `angle_radian` equal to 0):*

`tan(60)`

Output:

$$\sqrt{3}$$

- `cot(x)` returns the cotangent of x .

Examples.

* *Input (with angle_radian equal to 1):*

`cot(pi/6)`

Output:

$$\frac{2\sqrt{3}}{2}$$

* *Input (with angle_radian equal to 0):*

`cot(45)`

Output:

$$1$$

– `sec(x)` returns the secant of x .

Example.

* *Input (with angle_radian equal to 1):*

`sec(pi/3)`

Output:

$$2$$

* *Input (with angle_radian equal to 0):*

`sec(30)`

Output:

$$\frac{2}{\sqrt{3}}$$

– `csc(x)` returns the cosecant of x .

Examples.

* *Input (with angle_radian equal to 1):*

`csc(pi/4)`

Output:

$$\frac{2}{\sqrt{2}}$$

* *Input (with angle_radian equal to 0):*

`csc(30)`

Output:

$$2$$

- The `asin`, `acos`, `atan`, `acot`, `asec`, `acsc` commands are the inverse trigonometric functions. The latter are defined by:

– `asec(x)` = `acos(1/x)`,

– `acsc(x)` = `asin(1/x)`,

– `acot(x)` = `atan(1/x)`.

`arcsin` is a synonym for `asin`.

`arccos` is a synonym for `acos`.

`arctan` is a synonym for `atan`.

- These functions take one argument: x , a number.
They return a number which can represent an angle; by default, the angles will be in radians, but you can set `Xcas` to use degrees (see Section 3.5.3 p.71) by setting the variable `angle_radian` to 0; resetting it to 1 will change the angle measure to radians again.
- `asin(x)` returns the arcsine of x .

Examples.

* *Input (with `angle_radian` equal to 1):*

`asin(1/2)`

Output:

$$\frac{\pi}{6}$$

* *Input (with `angle_radian` equal to 0):*

`asin(1)`

Output:

$$\frac{\pi}{2}$$

- `acos(x)` returns the arccosine of x .

Examples.

* *Input (with `angle_radian` equal to 1):*

`acos(sqrt(3)/2)`

Output:

$$\frac{1}{6}\pi$$

* *Input (with `angle_radian` equal to 0):*

`acos(-1/2)`

Output:

$$120$$

- `atan(x)` returns the arctangent of x .

Examples.

* *Input (with `angle_radian` equal to 1):*

`atan(sqrt(3))`

Output:

$$\frac{\pi}{3}$$

* *Input (with `angle_radian` equal to 0):*

`atan(1)`

Output:

$$45$$

- `acot(x)` returns the arccotangent of x .

Examples.

* *Input (with `angle_radian` equal to 1):*

`acot(sqrt(3))`

Output:

$$\frac{\pi}{6}$$

* *Input (with angle_radian equal to 0):*

`acot(1/sqrt(3))`

Output:

60

– `asec(x)` returns the arcsecant of x .

Examples.

* *Input (with angle_radian equal to 1):*

`asec(1)`

Output:

0

* *Input (with angle_radian equal to 0):*

`asec(sqrt(2))`

Output:

45

– `acsc(x)` returns the arccosecant of x .

Examples.

* *Input (with angle_radian equal to 1):*

`acsc(1)`

Output:

$$\frac{\pi}{2}$$

* *Input (with angle_radian equal to 0):*

`acsc(2)`

Output:

30

- The `sinh`, `cosh`, and `tanh` commands compute the hyperbolic sine, hyperbolic cosine, and hyperbolic tangent functions.

– These functions take one argument:
 x , a number.

– `sinh(x)` returns the hyperbolic sine of x .

Example.

Input:

`sinh(1.0)`

Output:

1.17520119364

– `cosh(x)` returns the hyperbolic cosine of x .

Example.

Input:

`cosh(0)`

Output:

1

- `tanh(x)` returns the hyperbolic tangent of x .

Example.

Input:

`tanh(-1.0)`

Output:

-0.761594155956

- The `asinh`, `acosh`, and `atanh` commands compute the inverse hyperbolic functions.
`arcsinh` is a synonym for `asinh`.
`arccosh` is a synonym for `acosh`.
`arctanh` is a synonym for `atanh`.

- These functions take one argument:
 x , a number.
- `asinh(x)` returns the inverse hyperbolic sine of x .

Example.

Input:

`asinh(2)`

Output:

$\ln(2 + \sqrt{5})$

- `acosh(x)` returns the inverse hyperbolic cosine of x .

Example.

Input:

`acosh(1)`

Output:

0

- `atanh(x)` returns the inverse hyperbolic tangent of x .

Example.

Input:

`atanh(1/2)`

Output:

$\frac{\ln(3)}{2}$

6.16.3 Defining algebraic functions

Defining a function from \mathbb{R}^p to \mathbb{R}^q

If $expr$ is an expression possibly involving a variable x , you can use it to define a function f either by

$f(x) := expr$

or $f := x \rightarrow expr$

(see Section 5.5.1 p.109).

Warning!!!

The expression after \rightarrow is not evaluated. You should use **unapply** (see Section 6.15.2 p.235) if you expect the second member to be evaluated before the function is defined.

Example.

To define $f : (x) \rightarrow x * \sin(x)$,

Input:

```
f(x):=x*sin(x)
```

or:

```
f:=x->x*sin(x)
```

then:

```
f(pi/4)
```

Output:

$$\frac{\pi\sqrt{2}}{8}$$

You can similarly define a function of several variables, by replacing x by a sequence (x_1, \dots, x_p) or a list $[x_1, \dots, x_p]$ of variables.

Example.

Input:

```
f(x,y):=x*sin(y)
```

or:

```
f:=(x,y)->x*sin(y)
```

then:

```
f(2,pi/6)
```

Output:

1

You can also define a function with values in \mathbb{R}^q by replacing $expr$ by a sequence $(expr_1, \dots, expr_q)$ or list $[expr_1, \dots, expr_q]$ of expressions.

Examples.

- Define the function $h : (x, y) \rightarrow (x * \cos(y), x * \sin(y))$.

Input:

```
h(x,y):=(x*cos(y),x*sin(y))
```

then:

```
h(2,pi/4)
```

Output:

```
√2, √2
```

- Define the function $h(x, y) \rightarrow [x \cos(y), x \sin(y)]$.

Input:

```
h(x,y):=[x*cos(y),x*sin(y)];
```

or:

```
h:=(x,y)->[x*cos(y),x*sin(y)];
```

or:

```
h(x,y):={ [x*cos(y),x*sin(y)]};
```

or:

```
h:=(x,y)->return[x*cos(y),x*sin(y)];
```

or:

```
h(x,y):={return [x*cos(y),x*sin(y)]};
```

then:

```
h(2,pi/4)
```

Output:

```
[√2, √2]
```

Defining families of function from \mathbb{R}^{p-1} to \mathbb{R}^q using a function from \mathbb{R}^p to \mathbb{R}^q

Suppose that the function $f : (x, y) \rightarrow f(x, y)$ is defined, and you want to define a family of functions $g(t)$ such that $g(t)(y) := f(t, y)$ (i.e. t is viewed as a parameter). Since the expression after \rightarrow (or $:=$) is not evaluated, you should not define $g(t)$ by $g(t):=y->f(t,y)$; you have to use the `unapply` command (see Section 6.15.2 p.235).

For example, to define $f : (x, y) \rightarrow x \sin(y)$ and $g(t) : y \rightarrow f(t, y)$:

Input:

```
f(x,y):=x*sin(y);g(t):=unapply(f(t,y),y)
```

then:

$$g(2)$$

Output:

$$y \mapsto 2 \sin y$$

Input:

$$g(2)(1)$$

Output:

$$2 \sin(1)$$

For another example, suppose that you want to define the function $h : (x, y) \rightarrow [x * \cos(y), x * \sin(y)]$ and then you want to define the family of functions $k(t)$ having t as parameter such that $k(t)(y) := h(t, y)$. To define the function $h(x, y)$

Input:

$$h(x, y) := (x * \cos(y), x * \sin(y))$$

To define properly the function $k(t)$: *Input:*

$$k(t) := \text{unapply}(h(x, t), x)$$

then:

$$k(2)$$

Output:

$$x \mapsto (x \cos(2), x \sin(2))$$

$$(x) \rightarrow (x * \cos(2), x * \sin(2))$$

Input:

$$k(2)(1)$$

Output:

$$\cos(2), \sin(2)$$

6.16.4 Composing functions: @

With **Xcas**, the composition of functions is done with the infix operator **@** (see Section 6.14.1 p.229).

Examples.

- *Input:*

$$(sq@sin+id)(x)$$

Output:

$$\sin^2(x) + x$$

- *Input:*

$$(\sin@sin)(\pi/2)$$

Output:

$$\sin(1)$$

6.16.5 Repeated function composition: @@

With `Xcas`, the repeated composition of a function with itself $n \in \mathbb{N}$ times is done with the infix operator `@@` (see Section 6.14.1 p.229).

Examples.

- *Input:*

```
(sin@@3)(x)
```

Output:

```
sin(sin(sin x))
```

- *Input:*

```
(sin@@2)(pi/2)
```

Output:

```
sin(1)
```

6.16.6 Defining a function with history: as_function_of

The `as_function_of` command creates a function defined by an expression, even if the desired variable already has a value.

- `as_function_of` takes two arguments:
 - x , a variable.
 - `exprvar`, another variable containing an expression which itself may involve x .
- `as_function_of(exprvar, x)` returns a function defined by the expression that `exprvar` contains.

Example.

Input:

```
a:=sin(x)
```

Output:

```
sin(x)
```

Input:

```
b:=sqrt(1+a^2)
```

Output:

$$\sqrt{1 + \sin^2 x}$$

Input:

```
c:=as_function_of(b,a)
```

Output:

```
(a) -> { return(sqrt(1+a^2)); }
```

Input:

$c(x)$

Output:

$$\sqrt{1+x^2}$$

Warning !!

If the variable **b** has been assigned several times, the first assignment of **b** following the last assignment of **a** will be used. Moreover, the order used is the order of validation of the commandlines, which may not be reflected by the Xcas interface if you reused previous commandlines.

Example.

Input:

```
a:=2 b:=2*a+1 b:=3*a+2 c:=as_function_of(b,a)
```

Output:

```
(a) -> {return(2*a+1); }
```

So $c(x)$ is equal to $2x+1$. But: *Input:*

```
a:=2
b:=2*a+1
a:=2
b:=3*a+2
c:=as_function_of(b,a)
```

Output:

```
(a) -> {return(3*a+2); }
```

So $c(x)$ is equal to $3x+2$.

Hence the line where **a** is defined must be reevaluated before the good definition of **b**.

6.17 Getting information about functions from \mathbb{R} to \mathbb{R}

6.17.1 The domain of a function: domain

The **domain** command finds the domain of a function.

- **domain** takes one mandatory argument and one optional argument:
 - *expr*, an expression involving a single variable.
 - Optionally, *x*, the variable, which by default will be **x**.
- **domain**(*expr* $\langle x \rangle$) returns the domain of the function defined by *expr*.

Examples.

- *Input:*

$$\text{domain}(\ln(x+1))$$

Output:

$$x > -1$$

- *Input:*

$$\text{domain}(\text{asin}(2*t), t)$$

Output:

$$t \geq -\frac{1}{2} \wedge t \leq \frac{1}{2}$$

6.17.2 Table of variations of a function: tabvar

The table of variations of a function consists of

- The first row, for the variable, which gives the endpoints of subintervals of the domain, as well as any critical points and inflection points.
- The second row, for the derivative, which gives the values of the derivative at the values in the first row (or limits as the variable approaches one of the values) and between them the sign (+ or −) of the derivative in the corresponding subinterval.
- The third row, for the function, which gives the values of the function at the values in the first row, and between them whether the function is increasing or decreasing in the corresponding subinterval.
- The fourth row, for the second derivative, which gives the values of the second derivative at the values in the first row, and between them whether the second derivative is positive or negative (and hence whether the graph is concave up or concave down) in the subinterval.

The **tabvar** command finds the table of variations of a function.

- **tabvar** takes one mandatory argument and one optional argument.
 - *expr*, an expression of a single variable.
 - Optionally, *x*, the variable (by default, $x = x$).
- **tabvar**(*expr* $\langle, x \rangle$) returns the table of variations of the function $f(x) = \text{expr}$ and draws the graph on the **DispG** screen, accessible with the menu **Cfg►Show►DispG**.

Examples.

- *Input:*

```
tabvar(x^2 - x - 2,x)
```

Output:

```
Function plot x^2-x-2, variable x
Domain x
Vertical parabolic asymptote at -infinity
Vertical parabolic asymptote at +infinity
Variations x^2-x-2
```

$$\left[\begin{array}{cccccc} x & -\infty & & \frac{1}{2} & & +\infty \\ y' = 2x - 1 & -\infty & - & 0 & + & +\infty \\ y = x^2 - x - 2 & +\infty & \downarrow & -\frac{9}{4} & \uparrow & +\infty \\ y'' & 2 & +(\cup) & 2 & +(\cup) & 2 \end{array} \right]$$

```
plotfunc(x^2-x-2,x=(-0.6681472) .. 1.7222552))
Inside Xcas you can see the function with Cfg>Show>DispG.
```

• *Input:*

```
tabvar((2*t-1)/(t-1),t)
```

Output:

```
Function plot (2*t-1)/(t-1), variable t
Domain t<>1
Vertical asymptote x=1
Horizontal asymptote y=2
Horizontal asymptote y=2
Variations (2*t-1)/(t-1)
```

$$\left[\begin{array}{cccccc} t & -\infty & & 1 & 1 & +\infty \\ y' = -\frac{1}{(t-1)^2} & 0 & - & || & || & - & 0 \\ y = \frac{2t-1}{t-1} & 2 & \downarrow & -\infty & +\infty & \downarrow & 2 \\ y'' & 0 & -(\cap) & || & || & +(\cup) & 0 \end{array} \right]$$

```
plotfunc((2*t-1)/(t-1),t=(-0.1681472) .. 2.2222552))
Inside Xcas you can see the function with Cfg>Show>DispG.
```

Note that in this case, the value 1 appears twice in the first row, so that both one-sided limits of y can be displayed at the vertical asymptote $t = 1$. The values of 2 for y at $-\infty$ and ∞ indicate a horizontal asymptote of $y = 2$.

6.18 Limits: limit

The `limit` command computes limits, both at numbers and infinities, and in the real case it can compute one-sided limits.

- `limit` takes three mandatory and one optional argument.
 - *expr*, an expression.
 - *x*, the name of a variable.
 - *a*, the limit point.
 - Optionally, *side* (either 0, -1 or 1), to specify which side to take a one-sided limit (by default *side*=0).
- `limit(expr,x,a[,side])` returns the limit of *expr* as *x* approaches *a*.
 - If *side* is 0 (the default), then the ordinary limit is returned.
 - If *side* is -1, then the limit from the left ($x < a$) is returned.
 - If *side* is 1, then the limit from the right ($x > a$) is returned.

Remark:

It is also possible to put `x=a` as argument instead of `x,a`; `limit(expr,var=pt[,side])` is equivalent to `limit(expr,var,pt[,side])`.

Examples.

- *Input:*

```
limit(1/x,x,0,-1)
```

or:

```
limit(1/x,x=0,-1)
```

Output:

$-\infty$

- *Input:*

```
limit(1/x,x,0,1)
```

or:

```
limit(1/x,x=0,1)
```

Output:

$+\infty$

- *Input:*

```
limit(1/x,x,0,0)
```

or:

`limit(1/x,x,0)`

or:

`limit(1/x,x=0)`

Output:

∞

(Note that ∞ or *infinity* without an explicit + or - represents unsigned infinity.) Hence, `abs(1/x)` approaches $+\infty$ when x approaches 0.

Exercises.

- Find, for $n > 2$, the limit as x approaches 0 of:

$$\frac{n \tan(x) - \tan(nx)}{\sin(nx) - n \sin(x)}$$

Input:

`limit((n*tan(x)-tan(n*x))/(sin(n*x)-n*sin(x)),x=0)`

Output:

2

- Find the limit as x approaches $+\infty$ of

$$\sqrt{x + \sqrt{x + \sqrt{x}}} - \sqrt{x}$$

Input:

`limit(sqrt(x+sqrt(x+sqrt(x)))-sqrt(x),x=+infinity)`

Output:

$\frac{1}{2}$

- Find the limit as x approaches 0 of

$$\frac{\sqrt{1+x+x^2/2} - \exp(x/2)}{(1 - \cos(x)) \sin(x)}$$

Input:

`limit((sqrt(1+x+x^2/2)-exp(x/2))/((1-cos(x))*sin(x)),x,0)`

Output:

$-\frac{1}{6}$

6.19 Derivation and applications

6.19.1 Functional derivative: `function_diff`

The `function_diff` command finds the derivatives of functions (as opposed to expressions, see Section 6.15.1 p.234).

- `function_diff` takes one argument: f , a function.
- `function_diff(f)` returns the derivative f' of f .

Examples.

- *Input:*

```
function_diff(sin)
```

Output:

$$x \mapsto \cos x$$

- *Input:*

```
function_diff(sin)(x)
```

Output:

$$\cos x$$

- *Input:*

```
f(x):=x^2+x*cos(x)
function_diff(f)
```

Output:

$$x \mapsto \cos x - x \sin x + 2x$$

- *Input:*

```
function_diff(f)(x)
```

Output:

$$\cos x - x \sin x + 2x$$

- To define the function g as f' :

Input:

```
g:=function_diff(f)
```

- The `function_diff` instruction has the same effect as using the expression derivative `diff` (see Section 6.19.4 p.271) in conjunction with `unapply` (see Section 6.15.2 p.235):

Input:

```
g:=unapply(diff(f(x),x),x)
g(x)
```

Output:

$$\cos x - x \sin x + 2x$$

Warning!!!

In **Maple** mode (see Section 3.5.2 p.71), for compatibility, **D** may be used in place of **function_diff**. For this reason, it is impossible to assign a variable named **D** in **Maple** mode (hence you can not name a geometric object **D**).

6.19.2 Length of an arc: **arcLen**

The **arcLen** command finds the lengths of curves in the plane, which can either be given by an equation or a curve object.

To find the length of a curve given by an equation:

- **arcLen** takes four arguments:
 - *expr*, an expression (resp. a list of two expressions [*expr*₁, **expr**₂]) involving a variable *x*.
 - *x*, the name of the variable.
 - *a* and *b*, two values for the bounds of this variable.
- **arcLen**(*expr*, *x*, *a*, *b*) (resp. **arcLen**([*expr*₁, *expr*₂] *x*, *a*, *b*)) returns the length of the curve defined by $y = f(x) = \text{expr}$ (resp. by $x_1 = \text{expr}_1, x_2 = \text{expr}_2$) as *x* varies from *a* to *b*, using the formula

$$\text{arcLen}(f(x), x, a, b) = \int_a^b \sqrt{f'(x)^2 + 1} dx$$

or

$$\text{arcLen}(f(x), x, a, b) = \int_a^b \sqrt{x'(t)^2 + y'(t)^2} dt$$

Examples.

- Compute the length of the parabola $y = x^2$ from $x = 0$ to $x = 1$.

Input:

```
arcLen(x^2,x,0,1)
```

or:

```
arcLen([t,t^2],t,0,1)
```

Output:

$$\frac{2\sqrt{5} - \ln(\sqrt{5} - 2)}{4}$$

- Compute the length of the curve $y = \cosh(x)$ from $x = 0$ to $x = \ln(2)$.
- Input:*

```
arcLen(cosh(x),x,0,log(2))
```

Output:

$$\frac{3}{4}$$

- Compute the length of the circle $x = \cos(t)$, $y = \sin(t)$ from $t = 0$ to $t = 2 * \pi$.

Input:

```
arcLen([cos(t),sin(t)],t,0,2*pi)
```

Output:

$$2\pi$$

To find the length of a curve given by a curve object:

- `arcLen` takes a single argument: *curve*, a geometric curve defined in one of the graphics chapters (chapters 13 and 14).
- `arcLen(curve)` returns the length of the curve.

Examples.

- *Input:*

```
arcLen(circle(0,1,0,pi/2))
```

Output:

$$\frac{1}{2}\pi$$

- *Input:*

```
arcLen(arc(0,1,pi/2))
```

Output:

$$\frac{1}{4}\pi\sqrt{2}$$

6.19.3 Maximum and minimum of an expression: fMax fMin

The `fMax` and `fMin` commands find where maxima and minima occur. They can do this for expressions of one variable or for expressions of several variables subject to a set of constraints, either equalities or inequalities.

The find the maximum and minimum of an expression with one variable:

- `fMax` and `fMin` take two arguments:
 - *expr*, an expression involving one variable.
 - Optionally, *x*, the name of the variable (by default $x=\mathbf{x}$).
- `fMax(expr⟨,x⟩)` returns the value of *x* that maximizes the expression.

- `fMin(expr, x)` returns the value of x that minimizes the expression.

Examples.

- *Input:*

`fMax(sin(x), x)`

or:

`fMax(sin(x))`

or:

`fMax(sin(y), y)`

Output:

$$\frac{\pi}{2}$$

- *Input:*

`fMin(sin(x), x)`

or:

`fMin(sin(x))`

or:

`fMin(sin(y), y)`

Output:

$$-\frac{\pi}{2}$$

The find the maximum and minimum of an expression with several variables subject to constraints:

- `fMax` and `fMin` take four mandatory and two optional arguments:
 - *expr*, an expression with several variables.
 - *constr*, a list of constraints (equalities and inequalities).
 - *vars*, a list of the variables.
 - *init*, an initial guess (which must be a list of nonzero reals representing a feasible point).
 - Optionally, ϵ , the precision. If this isn't given, the default epsilon value is used (see Section 3.5.7 p.73, item 9).
 - Optionally, N , the maximum number of iterations.

The expression *expr* does not need to be differentiable.

- `fMax(expr, constr, vars, init ⟨, ϵ⟩ ⟨, N⟩)` returns the vector of values that maximizes *expr* subject to the constraints *constr*.
- `fMin(expr, constr, vars, init ⟨, ϵ⟩ ⟨, N⟩)` returns the vector of values that minimizes *expr* subject to the constraints *constr*.

Examples.

- *Input:*

```
fMax((x-2)^2+(y-1)^2, [-.25x^2-y^2+1>=0, x-2y+1=0], [x, y], [.5, .75])
```

Output:

```
[-1.82287565553, -0.411437827766]
```

- *Input:*

```
fMin((x-5)^2+y^2-25, [y>=x^2], [x, y], [1, 1])
```

Output:

```
[1.2347728625, 1.52466402196]
```

Although the initial point is required to be feasible, the algorithm will sometimes succeed even with a poor choice of initial point. Note that the initial value of a variable must not be zero.

6.19.4 Derivatives and partial derivatives

The `diff` command computes derivatives and partial derivatives of expressions.

`derive` is a synonym for `diff`.

To compute first order derivatives:

- `diff` takes one mandatory argument and one optional argument:
 - *expr*, an expression or a list of expressions.
 - Optionally, *x*, a variable (resp. a list of variable names, see several variable functions in 6.21). If the only variable is *x*, this second argument can be omitted.
- `diff(expr ⟨, x⟩)` returns the derivative (resp. a vector of derivatives) of the expression *expr* (or list of expressions) with respect to the variable *x* (resp. with respect to each variable in the list *x*).

Examples.

- Compute:

$$\frac{\partial(xy^2z^3 + xyz)}{\partial z}$$

Input:

`diff(x*y^2*z^3+x*y*z,z)`

Output:

$$3xy^2z^2 + xy$$

- Compute the 3 first order partial derivatives of $x * y^2 * z^3 + x * y * z$.

Input:

`diff(x*y^2*z^3+x*y,[x,y,z])`

Output:

$$[y^2z^3 + y, 2xyz^3 + x, 3xy^2z^2]$$

- Compute:

$$\frac{\partial^3(x.y^2.z^3 + x.y.z)}{\partial y \partial^2 z}$$

Input:

`diff(x*y ^2*z^3+x*y*z,y,z$2)`

Output:

$$12xyz$$

To compute higher order derivatives:

- `diff` takes more than two arguments:
 - *expr*, an expression.
 - x_1, x_2, \dots , the names of the derivation variables. Note that for repeated variables, you can use the `$` operator (see Section 6.39.2 p.454) followed by the number of derivations with respect to the variable; for example, instead of writing x, x, x you could write $x\$3$.
- `diff(expr, x1, x2, ...)` returns the partial derivative of *expr* with respect to the variables x_1, x_2, \dots

Examples.

- Compute:

$$\frac{\partial^2(xy^2z^3 + xyz)}{\partial x \partial z}$$

Input:

`diff(x*y^2*z^3+x*y*z,x,z)`

Output:

$$3y^2z^2 + y$$

- Compute:

$$\frac{\partial^3(xy^2z^3 + xyz)}{\partial x \partial^2 z}$$

Input:

```
diff(x*y^2*z^3+x*y*z,x,z,z)
```

or:

```
diff(x*y^2*z^3+x*y*z,x,z$2)
```

Output:

$$6y^2z$$

- Compute the third derivative of:

$$\frac{1}{x^2 + 2}$$

Input:

```
normal(diff((1)/(x^2+2),x,x,x))
```

or:

```
normal(diff((1)/(x^2+2),x$3))
```

Output:

$$\frac{-24x^3 + 48x}{x^8 + 8x^6 + 24x^4 + 32x^2 + 16}$$

Remark.

- Note the difference between `diff(f, x, y)` and `diff(f, [x, y])`:
`diff(f, x, y)` returns $\frac{\partial^2(f)}{\partial x \partial y}$ and `diff(f, [x, y])` returns $[\frac{\partial(f)}{\partial x}, \frac{\partial(f)}{\partial y}]$
- Never define a derivative function with `f1(x):=diff(f(x),x)`. Indeed, `x` would mean two different things Xcas is unable to deal with: on the left hand side, `x` is the variable name to define the f_1 function, and on the right hand side, `x` is the differentiation variable. The right way to define a derivative is either with `function_diff` or:

```
f1:=unapply(diff(f(x),x),x)
```

6.19.5 Implicit differentiation: `implicitdiff`

The `implicitdiff` command can differentiate implicitly defined functions or expressions containing implicitly defined functions. It has three different calling sequences.

To implicitly differentiate dependent variables:

- `implicitdiff` takes four arguments:
 - *constraints*, an equation or list of equations which implicitly define the dependent variables as functions of the independent variables; these will be of the form

$$g_i(x_1, \dots, x_n, y_1, \dots, y_m) = 0$$
 for $i = 1, 2, \dots, m$, where x_1, \dots, x_n are the independent variables and y_1, \dots, y_m are the dependent variables.
 - *depvars*, the list of dependent variables, where each dependent variable can optionally be written as a function of the x_i or the name written as a function of the independent variables $y_i(x_1, \dots, x_n)$. If there is only one dependent variable, this can be omitted.
 - *y*, a dependent variable or a list of dependent variables to be differentiated.
 - *diffvars*, a sequence of independent variables x_{i_1}, \dots, x_{i_k} with respect to differentiate.
- `implicitdiff(constraints [, depvars], y, diffvars)` returns the derivative (or list of derivatives) of *y* with respect to *diffvars*.

Examples.

- *Input:*

```
implicitdiff(x^2*y+y^2=1,y,x)
```

Output:

$$-\frac{2xy}{x^2 + 2y}$$

- *Input:*

```
implicitdiff([x^2+y=z,x+y*z=1],[y(x),z(x)],y,x)
```

Output:

$$\frac{-2xy - 1}{y + z}$$

To find a specified derivative of an expression containing implicitly defined functions:

- `implicitdiff` takes four arguments:

- *expr*, a differentiable expression involving independent variables x_1, x_2, \dots, x_n and dependent variables y_1, y_2, \dots, y_m .
- *constraints*, an equation or list of equations which implicitly define the dependent variables as functions of the independent variables; these will be of the form

$$g_i(x_1, \dots, x_n, y_1, \dots, y_m) = 0$$

for $i = 1, 2, \dots, m$.

- *depvars*, the dependent variable or list of dependent variables, where each dependent variable can either be the variable name y_i or the name written as a function of the independent variables $y_i(x_1, \dots, x_n)$.
- *diffvars*, a sequence of independent variables x_{i_1}, \dots, x_{i_k} with respect to which *expr* is differentiated.
- `implicitdiff(expr, implicitdef, depvars, diffvars)` returns the expression *expr* differentiated with respect to *diffvars*.

Example.

Input:

```
implicitdiff(x*y, -2x^3+15x^2*y+11y^3-24y=0, y(x), x)
```

Output:

$$\frac{2x^3 - 5x^2y + 11y^3 - 8y}{5x^2 + 11y^2 - 8}$$

To find all k th order derivatives of an expression involving implicitly defined functions:

- `implicitdiff` takes four mandatory arguments and one optional argument:
 - *expr*, a differentiable expression involving independent variables x_1, x_2, \dots, x_n and dependent variables y_1, y_2, \dots, y_m .
 - *constraints*, an equation or list of equations which implicitly define the dependent variables as functions of the independent variables; these will be of the form

$$g_i(x_1, \dots, x_n, y_1, \dots, y_m) = 0$$

for $i = 1, 2, \dots, m$.

- *vars*, a list $[x_1, \dots, x_n, y_1, \dots, y_m]$ of the independent and dependent variables entered as symbols in single list such that dependent variables come last.
- *order=k*, where k is the order of the derivatives to be taken.
- Optionally, *a*, a point where the partial derivatives should be evaluated at.

- `implicitdiff(expr, implicitdef, vars, order=k, a)` returns all partial derivatives of order k . If $k = 1$ they are returned in a single list, which represents the gradient of `expr` with respect to independent variables. If $k = 2$ the corresponding Hessian matrix is returned (see Section 6.21.3 p.293). If $k > 2$, a table with keys in form $[k_1, k_2, \dots, k_n]$, where $\sum_{i=1}^n k_i = k$, is returned. Such a key corresponds to

$$\frac{\partial^k f}{\partial var_1^{k_1} \partial var_2^{k_2} \dots \partial var_n^{k_n}}.$$

Examples.

- *Input:*

```
f:=x*y*z; g:=-2x^3+15x^2*y+11y^3-24y=0;
implicitdiff(f,g,[x,z,y],order=1)
```

Output:

$$\left[\frac{2x^3z - 5x^2yz + 11y^3z - 8yz}{5x^2 + 11y^2 - 8}, xy \right]$$

- *Input:*

```
implicitdiff(f,g,[x,z,y],order=2,[1,-1,0])
```

Output:

$$\begin{bmatrix} \frac{64}{9} & -\frac{2}{3} \\ -\frac{2}{3} & 0 \end{bmatrix}$$

- In the next example, the value of $\frac{\partial^4 f}{\partial x^4}$ is computed at the point $(x = 0, y = 0, z)$.

Input:

```
pd:=implicitdiff(f,g,[x,z,y],order=4,[0,z,0]);
pd[4,0]
```

Output:

$$-2z$$

6.19.6 Numerical differentiation: numdiff

The `numdiff` command finds numerical approximations to derivatives.

- `numdiff` takes three mandatory arguments and one optional argument.
 - $X = [\alpha_0, \alpha_1, \dots, \alpha_n]$, $Y = [\beta_0, \beta_1, \dots, \beta_n]$, two lists of real numbers, where $n \geq 1$.
 - x_0 , a real number.
 - Optionally, m , an integer or a sequence of integers (by default 1).

- `numdiff(X, Y, x0, m)` returns an approximation of the m -th derivative of a function f at x_0 , or a sequence of derivatives of order given by the sequence m , where f has values given by $f(\alpha_k) = \beta_k$, $k = 0, 1, \dots, n$.

`numdiff` uses Fornberg's algorithm described in "Generation of Finite Difference Formulas on Arbitrarily Spaced Grids", *Mathematics of Computation*, 51(184):699–706, 1988. The complexity of this algorithm is $O(n^2m)$ in both time and space. To avoid numerical instabilities, `numdiff` operates in exact arithmetic.

Note that $\alpha_0, \alpha_1, \dots, \alpha_n$ do not have to be equally spaced, but they must be mutually different and input in ascending order. There are no restrictions on the choice of x_0 .

Examples.

- Let $f(x) = \sin(x)e^{-x}$, $x \in [0, 1]$. Sample this function at the points in

$$X = [0, 0.1, 0.2, 0.4, 0.5, 0.7, 0.8, 1]$$

to approximate $f''(1/\pi)$.

Input:

```
f:=unapply(sin(x)*exp(-x),x)::
X:=[0,0.1,0.2,0.4,0.5,0.7,0.8,1]::
Y:=apply(f,X)::
```

Now you can approximate the second derivative of f at the point $x_0 = \frac{1}{\pi}$.

Input:

```
x0:=1/pi::
d:=numdiff(X,Y,x0,2)
```

Output:

$$-1.38167652799$$

Finally, compute the relative error of the obtained approximation.

Input:

$$\text{abs}(d - f''(x0)) / \text{abs}(f''(x0)) * 100$$

Output:

$$2.82975186496 \times 10^{-5}$$

The result is expressed in percentages.

- Use a sequence of values for the parameter m to find a list of approximations of the respective derivatives at x_0 . This is faster than calling `numdiff` to approximate one derivative at a time. Specifically, approximate the zeroth, first and second derivative of the function

$$f(x) = 1 - \frac{1}{1 + x^2}, \quad x \in [0, 1],$$

at the point $x_0 = \gamma$, where $\gamma \approx 0.57722$ is the Euler-Mascheroni constant, by sampling f at 21 equidistant points in the segment $[0, 1]$.

Input:

```
f:=unapply(1-1/(1+x^2),x)
X:=[(0.05*k)$(k=0..20)]; Y:=apply(f,X);
numdiff(X,Y,euler_gamma,0,1,2)
```

Output:

```
[0.249912571952, 0.649519026356, 0.000393517941567]
```

The correct values are $f(\gamma) = 0.249912571952$, $f'(\gamma) = 0.649519026356$ and $f''(\gamma) = 0.000393517946748$.

`numdiff` can be used for generating custom finite-difference stencils for approximation of derivatives.

Example.

Let $X = [-1, 0, 2, 4]$, $Y = [a, b, c, d]$ and $x_0 = 1$. To obtain an approximation formula for the second derivative:

Input:

```
numdiff([-1,0,2,4],[a,b,c,d],1,2)
```

Output:

$$\frac{2}{5}a - \frac{b}{2} + \frac{d}{10}$$

The approximation is always a linear combination of elements in Y , regardless of X , x_0 and m .

Given the lists $X = [\alpha_0, \alpha_1, \dots, \alpha_n]$ and $Y = [\beta_0, \beta_1, \dots, \beta_n]$, the Lagrange polynomial passing through points (α_k, β_k) where $k = 0, 1, \dots, n$ can be obtained by setting $m = 0$ and entering a symbol for x_0 .

Example.

Let $X = [-2, 0, 1]$ and $Y = [2, 4, 1]$:

Input:

```
expand(numdiff([-2,0,1],[2,4,1],x,0))
```

Output:

$$-\frac{4}{3}x^2 - \frac{5}{3}x + 4$$

The same result is obtained by entering `lagrange([-2,0,1],[2,4,1],x)`.

6.20 Integration

6.20.1 Antiderivative and definite integral: `integrate` `int` `Int`

The `int` and `integrate` commands compute a primitive or a definite integral. A difference between the two commands is that if you input `quest()` just after the evaluation of `integrate`, the answer is written with the \int symbol.

`Int` is the inert form of *integrate*; namely, it evaluates to *integrate* for later evaluation.

To find a primitive (an antiderivative):

- `int` (or `integrate`) takes one mandatory argument and one optional argument:
 - *expr*, an expression.
 - Optionally, *x*, the name of a variable (by default the value is `x`, so if the variable is `x` the second argument is unnecessary).
- `int(expr ⟨, x⟩)` (or `integrate(expr ⟨, x⟩)`) returns a primitive of *expr* with respect to *x*.

Examples.

- *Input:*

```
integrate(x^2)
```

Output:

$$\frac{x^3}{3}$$

- *Input:*

```
integrate(t^2,t)
```

Output:

$$\frac{t^3}{3}$$

To evaluate a definite integral:

- `int` (or `integrate`) takes four arguments:
 - *expr*, an expression.
 - *x*, the variable.
 - *a* and *b*, the bounds of the definite integral.
- `int(expr, x, a, b)` (or `integrate(expr, x, a, b)`) returns the exact value of the definite integral if the computation was successful or an unevaluated integral otherwise.

Examples.

- *Input:*

```
integrate(x^2,x,1,2)
```

Output:

$$\frac{7}{3}$$

• *Input:*

```
integrate(1/(sin(x)+2),x,0,2*pi)
```

Output:

$$\frac{2}{3}\pi\sqrt{3}$$

`Int` is the inert form of `integrate`, it prevents evaluation, for example to avoid a symbolic computation that might not be successful if you just want a numeric evaluation.

Example.

Input:

```
evalf(Int(exp(x^2),x,0,1))
```

or:

```
evalf(int(exp(x^2),x,0,1))
```

Output:

```
1.46265174591
```

Exercises.

1. Let

$$f(x) = \frac{x}{x^2 - 1} + \ln\left(\frac{x+1}{x-1}\right)$$

Find a primitive of f .

Input:

```
int(x/(x^2-1)+ln((x+1)/(x-1)))
```

Output:

$$x \ln\left(\frac{x+1}{x-1}\right) + \frac{2}{2} \ln|x^2 - 1| + \frac{\ln|x^2 - 1|}{2}$$

Alternatively, define the function `f`,

Input:

```
f(x):=x/(x^2-1)+ln((x+1)/(x-1))
```

then:

```
int(f(x))
```

The output, of course, will be the same.

Warning.

For `Xcas`, `log` is the natural logarithm (like `ln`); `log10` is the base-10 logarithm.

2. Compute:

$$\int \frac{2}{x^6 + 2 \cdot x^4 + x^2} dx$$

Input:

```
int(2/(x^6+2*x^4+x^2))
```

Output:

$$2 \left(\frac{-3x^2 - 2}{2(x^3 + x)} - \frac{3}{2} \arctan x \right)$$

3. Compute:

$$\int \frac{1}{\sin(x) + \sin(2 \cdot x)} dx$$

Input:

```
integrate(1/(sin(x)+sin(2*x)))
```

Output:

$$2 \left(\frac{\ln \left(\frac{1 - \cos x}{1 + \cos x} \right)}{12} - \frac{\ln \left| \frac{1 - \cos x}{1 + \cos x} - 3 \right|}{3} \right)$$

6.20.2 Primitive and definite integral: `risch`

The Risch algorithm is a powerful algorithm for finding an elementary primitive of an elementary function or concluding that one doesn't exist. The `risch` command finds primitives and can use them to evaluate definite integrals.

To find a primitive:

- `risch` takes one mandatory argument and one optional argument:
 - `expr`, an expression.
 - Optionally `x`, the name of a variable (by default the variable is `x`).
- `risch(expr <, x>)` returns a primitive of `expr` with respect to `x`.

Examples.

- *Input:*

```
risch(x^2)
```

Output:

$$\frac{x^3}{3}$$

- *Input:*

`risch(t^2,t)`

Output:

$$\frac{t^3}{3}$$

- *Input:*

`risch(exp(-x^2))`

Output:

$$\int e^{-x^2} dx$$

meaning that $\exp(-x^2)$ has no primitive expressed with the usual functions.

To evaluate a definite integral:

- `risch` takes four arguments:
 - *expr*, an expression *expr*.
 - *x*, the variable.
 - *a* and *b*, the bounds of the definite integral.
- `int(expr,x,a,b)` returns the exact value of the definite integral if the computation was successful or an unevaluated integral otherwise.

Example.

Input:

`risch(x^2,x,0,1)`

Output:

$$\frac{1}{3}$$

6.20.3 Discrete summation: `sum`

The `sum` command can evaluate sums, series, and find discrete antiderivatives. A discrete antiderivative of a sum $\sum_n f(n)$ is an expression G such that $G_{|x=n+1} - G_{|x=n} = f(n)$, which means that $\sum_{n=M}^N f(n) = G_{|x=N+1} - G_{|x=M}$. To evaluate a sum or series:

- `sum` takes four arguments:
 - *expr*, an expression.
 - *k*, the name of the variable.
 - n_0 and n_1 , integers (the bounds of the sum).
- `sum(expr,k,n0,n1)` returns the sum $\sum_{k=n_0}^{n_1} expr$.

Examples.

- *Input:*

$$\text{sum}(1, k, -2, n)$$

Output:

$$n + 1 + 2$$

- *Input:*

$$\text{normal}(\text{sum}(2*k-1, k, 1, n))$$

Output:

$$n^2$$

- *Input:*

$$\text{sum}(1/(n^2), n, 1, 10)$$

Output:

$$\frac{1968329}{1270080}$$

- *Input:*

$$\text{sum}(1/(n^2), n, 1, +(\text{infinity}))$$

Output:

$$\frac{1}{6}\pi^2$$

- *Input:*

$$\text{sum}(1/(n^3-n), n, 2, 10)$$

Output:

$$\frac{27}{110}$$

- *Input:*

$$\text{sum}(1/(n^3-n), n, 2, +(\text{infinity}))$$

Output:

$$\frac{1}{4}$$

This result comes from the decomposition of $1/(n^3-n)$ (see Section 6.32.9 p.416).

Input:

$$\text{partfrac}(1/(n^3-n))$$

Output:

$$-\frac{1}{n} + \frac{1}{2(n-1)} + \frac{1}{2(n+1)}$$

Hence:

$$\begin{aligned} \sum_{n=2}^N -\frac{1}{n} &= -\sum_{n=1}^{N-1} \frac{1}{n+1} = -\frac{1}{2} - \sum_{n=2}^{N-2} \frac{1}{n+1} - \frac{1}{N} \\ \frac{1}{2} \sum_{n=2}^N \frac{1}{n-1} &= \frac{1}{2} \left(\sum_{n=0}^{N-2} \frac{1}{n+1} \right) = \frac{1}{2} \left(1 + \frac{1}{2} + \sum_{n=2}^{N-2} \frac{1}{n+1} \right) \\ \frac{1}{2} \sum_{n=2}^N \frac{1}{n+1} &= \frac{1}{2} \left(\sum_{n=2}^{N-2} \frac{1}{n+1} + \frac{1}{N} + \frac{1}{N+1} \right) \end{aligned}$$

After simplification by $\sum_{n=2}^{N-2}$, it remains:

$$-\frac{1}{2} + \frac{1}{2} \left(1 + \frac{1}{2} \right) - \frac{1}{N} + \frac{1}{2} \left(\frac{1}{N} + \frac{1}{N+1} \right) = \frac{1}{4} - \frac{1}{2N(N+1)}$$

Therefore:

- for $N = 10$ the sum is equal to: $1/4 - 1/220 = 27/110$
- for $N = +\infty$ the sum is equal to: $1/4$ because $\frac{1}{2N(N+1)}$ approaches zero when N approaches infinity.

To find a discrete antiderivative:

- **sum** takes two arguments:
 - *expr*, an expression.
 - *k*, the name of the variable.
- **sum(expr,x)** returns a discrete antiderivative.

Example.

Input:

```
sum(1/(x*(x+1)),x)
```

Output:

$$-\frac{1}{x}$$

6.20.4 Riemann sum: sum_riemann

Given a function f on $[0, 1]$, the Riemann sum corresponding to dividing the interval into n equal parts and using the right endpoints is

$$\sum_{k=1}^n f\left(\frac{x}{n}\right) \frac{1}{n}.$$

The **sum_riemann** command determines if a sum is such a Riemann sum, and if it is, evaluates the integral.

- **sum_riemann** takes two arguments:

- $expr$, an expression depending on two variables.
- $[n, k]$, the list of those two variables.

- `sum_riemann(expr, [n, k])` returns

$$\lim_{n \rightarrow \infty} \sum_{k=1}^n expr$$

(which, viewing the sum as a Riemann sum of a continuous function on $[0, 1]$, is the definite integral) or returns "it is probably not a Riemann sum" when the no result is found.

Exercises.

1. Suppose $S_n = \sum_{k=1}^n \frac{k^2}{n^3}$.
Compute $\lim_{n \rightarrow +\infty} S_n$.

Input:

`sum_riemann(k^2/n^3, [n, k])`

Output:

$$\frac{1}{3}$$

2. Suppose $S_n = \sum_{k=1}^n \frac{k^3}{n^4}$.
Compute $\lim_{n \rightarrow +\infty} S_n$.

Input:

`sum_riemann(k^3/n^4, [n, k])`

Output:

$$\frac{1}{4}$$

3. Compute $\lim_{n \rightarrow +\infty} \left(\frac{1}{n+1} + \frac{1}{n+2} + \dots + \frac{1}{n+n} \right)$.

Input:

`sum_riemann(1/(n+k), [n, k])`

Output:

$$\ln(2)$$

4. Suppose $S_n = \sum_{k=1}^n \frac{32n^3}{16n^4 - k^4}$.
Compute $\lim_{n \rightarrow +\infty} S_n$.

Input:

`sum_riemann(32*n^3/(16*n^4-k^4), [n, k])`

Output:

$$2 \arctan\left(\frac{1}{2}\right) + \ln(3)$$

6.20.5 Integration by parts

Recall the integration by parts formula:

$$\int u(x)v'(x)dx = u(x)v(x) - \int v(x)u'(x)dx.$$

If you want to integrate a function $f(x)$ by parts, you need to specify how to write $f(x)$ as $u(x)v'(x)$, which you can do by either specifying $u(x)$ or $v(x)$. The result will be in the form $F(x) + \int g(x)dx$, where $F(x) = u(x)v(x)$ and $g(x) = -v(x)u'(x)$.

In some cases, to finish an integral you need to integrate by parts more than once. After one integrating by parts once and getting $F(x) + \int g(x)dx$, you may have to integrate $\int g(x)dx$ by parts and add $F(x)$ to the result.

Xcas has two commands for integrating by parts: **ibpdv** (where you specify $v(x)$) and **ibpu** (where you specify $u(x)$), both of which return the result as a list $[F(x), g(x)]$. Both of these commands allow you to keep track of the function $F(x)$ you may need to add to the result of a subsequent integration by parts.

ibpdv

The **ibpdv** command is used to search the primitive of an expression written as $u(x)v'(x)$ by specifying $v(x)$.

- **ibpdv** takes two arguments:
 - *uvprime*, an expression which you can think of as $u(x)v'(x)$, or $[Fexpr, uvprime]$, a list of two expressions, where again you can think of *uvprime* as $u(x)v'(x)$, and *Fexpr* represents the function $F(x)$ that you can add to the result of integrating by parts.
 - *vexpr*, an expression you can think of as $v(x)$. If *vexpr* is 0, then instead of integrating by parts, the expression *uvprime* is integrated as a whole (this can be useful for finishing a multi-step integration by parts problem).
- **ibpdv**(*uvprime*, *vexpr*) (or **ibpdv**($[Fexpr, uvprime], vexpr$)) returns:
 - If *vexpr* is not 0:
 - $[u(x)v(x), -v(x)u'(x)]$ (or $[F(x) + u(x)v(x), -v(x)u'(x)]$ if the first argument is a list).
 - If *vexpr* is 0:
 - $G(x)$ (or $F(x) + G(x)$, if the first argument is a list), where $G(x)$ is a primitive of *uvprime*.

Hence, **ibpdv** returns the terms computed in an integration by parts, with the possibility of doing several **ibpdv**s successively.

When the answer of **ibpdv**($u(x)*v'(x), v(x)$) is computed, to obtain a primitive of $u(x)v'(x)$, it remains to compute the integral of the second term of this answer and then to sum this integral with the first term of this answer:

to do this, just use `ibpdv` command with the answer as first argument and a new $v(x)$ (or 0 to terminate the integration) as second argument.

Example.

Input:

```
ibpdv(ln(x),x)
```

Output:

```
[x ln x, -1]
```

then:

```
ibpdv([x*ln(x), -1], 0)
```

or:

```
ibpdv(ans(), 0)
```

Output:

```
-x + x ln x
```

Remark.

When the first argument of `ibpdv` is a list of two elements, `ibpdv` works only on the last element of this list and adds the integrated term to the first element of this list. (therefore it is possible to do several `ibpdvs` successively).

Example. To evaluate $\int (\ln(x))^2 dx$:

Input:

```
ibpdv((ln(x))^2, x)
```

Output:

```
[x ln^2 x, -2 ln x]
```

It remains to integrate $-(2*\ln(x))$:

Input:

```
ibpdv([x*(ln(x))^2, -(2*log(x))], x)
```

or:

```
ibpdv(ans(), x)
```

Output:

```
[x ln^2 x - 2x ln x, 2]
```

And now it remains to integrate 2:

Input:

```
ibpdv([x*(ln(x))^2+x*(-(2*log(x))), 2], 0)
```

or:

```
ibpdv(ans(), 0)
```

Output:

```
x ln^2 x - 2x ln x + 2x
```

ibpu

The **ibpu** command is used to search the primitive of an expression written as $u(x)v'(x)$ by specifying $u(x)$.

- **ibpu** takes two arguments:
 - *uvprime*, an expression which you can think of as $u(x)v'(x)$, or $[Fexpr, uvprime]$, a list of two expressions, where again you can think of *uvprime* as $u(x)v'(x)$, and *Fexpr* represents the function $F(x)$ that you can add to the result of integrating by parts.
 - *uexpr*, an expression you can think of as $u(x)$. If *uexpr* is 0, then instead of integrating by parts, the expression *uvprime* is integrated as a whole (this can be useful for finishing a multi-step integration by parts problem).
- **ibpu**(*uvprime*, *uexpr*) (or **ibpu**($[Fexpr, uvprime]$, *uexpr*)) returns:
 - If **uexpr** is not 0:

$$[u(x)v(x), -v(x)u'(x)]$$
 (or $[F(x) + u(x)v(x), -v(x)u'(x)]$ if the first argument is a list).
 - If **uexpr** is 0:

$$G(x)$$
 (or $F(x) + G(x)$, if the first argument is a list), where $G(x)$ is a primitive of *uvprime*.

Hence, **ibpu** returns the terms computed in an integration by parts, with the possibility of doing several **ibpus** successively.

When the answer of **ibpu**(**u(x)*v'(x)**, **u(x)**) is computed, to obtain a primitive of $u(x)v'(x)$, it remains to compute the integral of the second term of this answer and then to sum this integral with the first term of this answer: to do this, just use the **ibpu** command with the answer as first argument and a new $u(x)$ (or 0 to terminate the integration) as second argument.

Example.

Input:

$$\text{ibpu}(\ln(x), \ln(x))$$

Output:

$$[x \ln x, -1]$$

then:

$$\text{ibpu}([x \ln(x), -1], 0)$$

or:

$$\text{ibpu}(\text{ans}(), 0)$$

Output:

$$-x + x \ln x$$
Remark.

When the first argument of **ibpu** is a list of two elements, **ibpu** works only on

the last element of this list and adds the integrated term to the first element of this list. Therefore it is possible to do several `ibpus` successively, similarly to how you can do several `ibpds` successively.

Example.

To evaluate $\int (\ln(x))^2 dx$:

Input:

```
ibpu((ln(x))^2, (ln(x))^2)
```

Output:

$$[x \ln^2 x, -2 \ln x]$$

It remains to integrate $-(2 \ln(x))$:

Input:

```
ibpu([x*(ln(x))^2, -(2*ln(x))], ln(x))
```

or:

```
ibpu(ans(), ln(x))
```

Output:

$$[x \ln^2 x - 2x \ln x, 2]$$

Finally, it remains to integrate 2: *Input:*

```
ibpu([x*(ln(x))^2+x*(-(2*ln(x))), 2], 0)
```

or:

```
ibpu(ans(), 0)
```

Output:

$$x \ln^2 x - 2x \ln x + 2x$$

6.20.6 Change of variables: `subst`

See the `subst` command in Section 6.12.18 p.216.

6.20.7 Integrals and limits

The `limit` command (see Section 6.18 p.265) can compute limits involving integrals.

Examples.

- Find the limit, as a approaches $+\infty$, of

$$\int_2^a \frac{1}{x^2} dx$$

Input (if a is assigned, first input `purge(a)`):

```
limit(integrate(1/(x^2), x, 2, a), a, +(infinity))
```

Output:

$$\frac{1}{2}$$

Since $\int_2^a 1/x^2 dx = 1/2 - 1/a$, the integral $\int_2^a 1/x^2 dx$ tends to $1/2$ as a goes to infinity.

- Find the limit, as a approaches $+\infty$, of

$$\int_2^a \left(\frac{x}{x^2-1} + \ln \left(\frac{x+1}{x-1} \right) \right) dx$$

Input (if a is assigned, first input `purge(a)`):

```
limit(integrate(x/(x^2-1)+log((x+1)/(x-1)),x,2,a),a,+infinity)
```

Output:

$$+\infty$$

Since $\int_2^a x/(x^2-1) dx = (1/2)(\ln(a^2-1) - \ln(3))$ and $\int_2^a \ln((x+1)/(x-1)) dx = \ln(a+1) + \ln(a-1) + a \ln((a+1)/(a-1)) - 3 \ln(3)$, the integral $\int_2^a x/(x^2-1) + \ln((x+1)/(x-1)) dx$ goes to infinity as a goes to infinity.

- For an example when the integral can't be simply evaluated, find the limit, as a approaches 0, of

$$\int_a^{3a} \frac{\cos(x)}{x} dx$$

Input:

```
limit(int(cos(x)/x,x,a,3a),a,0)
```

Output:

$$\ln(3)$$

To find this limit yourself, you can note that $1 - x^2/2 \leq \cos(x) \leq 1$, and so $1/x - x/2 \leq \cos(x)/x \leq 1/x$, and so $\int_a^{3a} 1/x - x/2 dx \leq \int_a^{3a} \cos(x)/x dx \leq \int_a^{3a} 1/x dx$, which gives you $\ln(3) - 2a^2 \leq \int_a^{3a} \cos(x)/x dx \leq \ln(3)$, and so as a approaches 0, $\int_a^{3a} \cos(x)/x dx$ will approach $\ln(3)$.

6.21 Multivariate calculus

6.21.1 Gradient: `derive` `deriver` `diff` `grad`

The `derive` command finds partial derivatives of a multivariable expression. `diff` and `grad` can be used synonymously for `derive` here.

- `derive` takes two arguments:
 - `expr`, an expression involving n real variables.
 - $[x_1, \dots, x_n]$, a vector of the variable names.

- `derive(expr, [x1, ..., xn])` returns the gradient of *expr*; namely, the vector of partial derivatives of *expr* with respect to x_1, \dots, x_n .

For example, in dimension $n = 3$, with variables $[x, y, z]$,

$$\overrightarrow{\text{grad}}(F) = \left[\frac{\partial F}{\partial x}, \frac{\partial F}{\partial y}, \frac{\partial F}{\partial z} \right]$$

Example.

Find the gradient of $F(x, y, z) = 2x^2y - xz^3$.

Input:

```
derive(2*x^2*y-x*z^3, [x,y,z])
```

or:

```
diff(2*x^2*y-x*z^3, [x,y,z])
```

or:

```
grad(2*x^2*y-x*z^3, [x,y,z])
```

Output:

$$[2 \cdot 2xy - z^3, 2x^2, -3xz^2]$$

Output after simplification with `normal(ans())`:

$$[4xy - z^3, 2x^2, -3xz^2]$$

To find the critical points of $F(x, y, z) = 2x^2y - xz^3$:

Input:

```
solve(derive(2*x^2*y-x*z^3, [x,y,z]), [x,y,z])
```

Output:

$$[[0, y, 0]]$$

6.21.2 Laplacian: laplacian

Recall, the Laplacian of a function F of n variables x_1, \dots, x_n is

$$\nabla^2(F) = \frac{\partial^2 F}{\partial x_1^2} + \frac{\partial^2 F}{\partial x_2^2} + \cdots + \frac{\partial^2 F}{\partial x_n^2}$$

Also, the $n \times n$ discrete Laplacian matrix (also called the second difference matrix) is the $n \times n$ tridiagonal matrix with 2s on the main diagonal, -1 s just above and below the main diagonal;

$$\begin{pmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & -1 & 2 & -1 \\ 0 & \cdots & 0 & -1 & 2 \end{pmatrix}$$

If L is the $n \times n$ discrete Laplacian matrix and Y is an $n \times 1$ column vector whose k th coordinate is $y_i = y(a + k\Delta x)$ for a twice differential function y , then the k th coordinate of LY will be $-y(a + (k-1)\Delta x) + 2y(a + k\Delta x) - y(a + (k+1)\Delta x)$ (implicitly assuming that $y(a) = y(a + (N+1)\Delta x) = 0$), which approximates $y''(a + k\Delta x)$. So LY is approximately $-\Delta x^2 Y''$, where Y'' is the $n \times 1$ column vector whose k th coordinate is $y''(a + k\Delta x)$.

The `laplacian` command can compute the Laplacian operator or the discrete Laplacian matrix.

To compute the Laplacian operator:

- `laplacian` takes two arguments:
 - *expr*, an expression involving several variables.
 - *vars*, a list of the variable names.
- `laplacian(expr,vars)` returns the Laplacian of the expression.

Example

Find the Laplacian of $F(x, y, z) = 2x^2y - xz^3$.

Input:

```
laplacian(2*x^2*y-x*z^3,[x,y,z])
```

Output:

$$-6xz + 4y$$

To compute the discrete Laplacian matrix:

- `laplacian` takes one argument:
 - n , an integer or floating-point integer.
- `laplacian(n)` returns the $n \times n$ discrete Laplacian matrix.

Examples.

- *Input:*

```
laplacian(3)
```

Output:

$$\begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{bmatrix}$$

- *Input:*

```
laplacian(2.0)
```

Output:

$$\begin{bmatrix} 2.0 & -1.0 \\ -1.0 & 2.0 \end{bmatrix}$$

6.21.3 Hessian matrix: `hessian`

Recall, the Hessian of a function F of n variables x_1, \dots, x_n is the matrix of second order derivatives:

$$\begin{pmatrix} \frac{\partial^2 F}{\partial x_1^2} & \cdots & \frac{\partial^2 F}{\partial x_1 \partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 F}{\partial x_n \partial x_1} & \cdots & \frac{\partial^2 F}{\partial x_n^2} \end{pmatrix}$$

The `hessian` command computes the Hessian of a function.

- `hessian` takes two arguments:
 - *expr*, an expression involving several variables.
 - *vars*, a list of the variable names.
- `hessian(expr, vars)` returns the Hessian of the expression.

Examples.

- Find the Hessian matrix of $F(x, y, z) = 2x^2y - xz^3$.
Input:

```
hessian(2*x^2*y-x*z^3 , [x,y,z])
```

Output:

$$\begin{bmatrix} 4y & 4x & -3z^2 \\ 2 \cdot 2x & 0 & 0 \\ -3z^2 & 0 & -2 \cdot 3xz \end{bmatrix}$$

- To get the Hessian matrix at the critical points:
Input:

```
solve(derive(2*x^2*y-x*z^3, [x,y,z]), [x,y,z])
```

Output (the critical points):

$$[[0, y, 0]]$$

Input (to evaluate the Hessian at these points):

```
subst([ [4*y, 4*x, -(3*z^2)], [2*2*x, 0, 0],  
      [-(3*z^2), 0, 6*x*z] ], [x,y,z], [0,y,0])
```

Output:

$$\begin{bmatrix} 4y & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

6.21.4 Divergence: divergence

Recall that the divergence of a vector field $\mathbf{F} = [F_1, \dots, F_n]$ with variables $[x_1, \dots, x_n]$ is

$$\operatorname{div} \mathbf{F} = \frac{\partial F_1}{\partial x_1} + \dots + \frac{\partial F_n}{\partial x_n}$$

The **divergence** command computes the divergence of a vector field.

- **divergence** takes two arguments:
 - F , a vector field given as a list $[F_1, \dots, F_n]$ of expressions.
 - $vars$, a list of the variable names.
- **divergence**($F, vars$) returns the divergence of the vector field F .

Example.

Input:

```
divergence([x*z, -y^2, 2*x*y], [x, y, z])
```

Output:

$-2y + z$

6.21.5 Rotational: curl

The curl of a three-dimensional vector field $\mathbf{F} = [F_1, F_2, F_3]$ with variables $[x_1, x_2, x_3]$ is

$$\operatorname{curl} \mathbf{F} = \left[\frac{\partial F_3}{\partial x_2} - \frac{\partial F_2}{\partial x_3}, \frac{\partial F_1}{\partial x_3} - \frac{\partial F_3}{\partial x_1}, \frac{\partial F_2}{\partial x_1} - \frac{\partial F_1}{\partial x_2} \right]$$

The **curl** command computes the curl of a three dimensional vector field (note that it must be three dimensional).

- **curl** takes two arguments:
 - F , a three-dimensional vector field, given as a list of three expressions depending on three variables.
 - $vars$, a list of the three variable names.
- **curl**($F, vars$) returns the curl of the vector field.

Example.

Input:

```
curl([x*z, -y^2, 2*x*y], [x, y, z])
```

Output:

$[2 \ln x \cdot x^y, x - 2yx^{y-1}, 0]$

6.21.6 Potential: potential

Recall that a vector field \mathbf{F} is conservative if there is a scalar-valued function f such that $\text{grad}f = \mathbf{F}$. In this case, f is called a potential of \mathbf{F} , and is only determined up to a constant.

The `potential` command computes the potential of a vector field, or signals an error if the vector field is not conservative.

- `potential` takes two arguments:
 - \mathbf{F} , a vector field given as a list of n expressions involving n variables.
 - $vars$, a list of the variable names.
- `potential($\mathbf{F}, vars$)` returns a potential function for \mathbf{F} if \mathbf{F} is conservative, and raises an error otherwise.

Note that `potential` is the reciprocal function of `derive`.

Example.

Input:

```
potential([2*x*y+3,x^2-4*z,-4*y],[x,y,z])
```

Output:

$$x^2y + 3x - 4yz$$

Note that in \mathbb{R}^3 , a vector field \mathbf{F} is conservative if and only if its curl is zero; i.e., if $\text{curl}\mathbf{F} = \mathbf{0}$. In time-independent electro-magnetism, $\mathbf{F} = \mathbf{E}$ is the electric field and f is the electric potential.

6.21.7 Conservative flux field: vpotential

A vector field \mathbf{F} in \mathbb{R}^3 is a conservative flux field, or a solenoidal field, if there is a vector field \mathbf{G} such that $\text{curl}\mathbf{G} = \mathbf{F}$. Given a conservative flux vector field \mathbf{F} , the general solution of $\text{curl}\mathbf{G} = \mathbf{F}$ is the sum of a particular solution and the gradient of an arbitrary functions.

The `vpotential` command finds a particular vector field \mathbf{G} such that $\text{curl}\mathbf{G} = \mathbf{F}$ if \mathbf{F} is a conservative flux field, and signals an error otherwise. Specifically, `vpotential` returns the solution \mathbf{G} with zero as the first component.

- `vpotential` takes two arguments:
 - F , a vector field in \mathbb{R}^3 , given as a list of three expressions depending on three variables.
 - $vars$, a list of the variable names.
- `vpotential($Fvars$)` returns a solution of $\text{curl}\mathbf{G} = \mathbf{F}$ whose first coordinate is zero if \mathbf{F} is a conservative vector field, and signals an error otherwise.

`vpotential` is the reciprocal function of `curl`.

Example.

Input:

```
vpotential([2*x*y+3,x^2-4*z,-2*y*z],[x,y,z])
```

Output:

$$\left[0, -2xyz, -\frac{x^3}{3} + 4xz + 3y\right]$$

In \mathbb{R}^3 , a vector field \mathbf{F} is a curl if and only if its divergence is zero. In time-independent electro-magnetism, $\mathbf{F} = \mathbf{B}$ is the magnetic field and $\mathbf{G} = \mathbf{A}$ is the potential vector.

6.21.8 Determining where a function is convex: `convex`

The `convex` command determines where a function is convex.

- `convex` takes two mandatory arguments and one optional argument:
 - *expr*, an expression which is at least twice differentiable, which specifies a function f .
 - *vars*, the variable or list of variables in the expression. Some variables may depend on a common independent parameter, say t , when entered as e.g. $x(t)$ instead of x . The first derivatives of such variables, when encountered in f , are treated as independent parameters of the function.
 - Optionally, `simplify=bool`, where *bool* can be `true` or `false`.
- `convex(expr,vars[,simplify=bool])` returns:
 - `true`, if the function is convex on the entire domain.
 - `false`, if the function is nowhere convex.
 - otherwise, the region where the function is convex is returned as inequalities (not necessarily independent) involving the variables.

The command operates by computing the Hessian H_f of f (see Section 6.21.3 p.293) and its principal minors (in total 2^n of them where n is the number of parameters) and checks their signs. If all minors are nonnegative, then H_f is positive semidefinite and f is therefore convex. Simplification is by default applied when generating convexity conditions. With a third argument of `simplify=false`, only rational normalization is performed (using the `ratnormal` command). `simplify=true` is the same as the default.

The function f is said to be *concave* if the function $g = -f$ is convex.

Examples.

- *Input:*

```
convex(3*exp(x)+5x^4-ln(x),x)
```


Output:

true

- *Input:*

`convex(x^2+y^2+3z^2-x*y+2x*z+y*z, [x,y,z])`

Output:

true

- *Input:*

`convex(x1^3+2x1^2+2*x1*x2+x2^2/2-8x1-2x2-8, [x1,x2])`

Output:

$[x_1 \geq 0]$

- In the example below, the function $f(x, y, z) = x^2 + xz + ayz + z^2$ is not convex regardless of the value $a \in \mathbb{R}$:

Input:

`convex(x^2+x*z+a*y*z+z^2, [x,y,z])`

Output:

false

- For the next example, find all values $a \in \mathbb{R}$ for which the function

$$f(x, y, z) = x^2 + 2y^2 + az^2 - 2xy + 2xz - 6yz$$

is convex on \mathbb{R}^3 .

Input:

`convex(x^2+2y^2+a*z^2-2x*y+2x*z-6y*z, [x,y,z])`

Output:

$[a \geq 5]$

Therefore f is convex for $a \geq 5$.

- Find the set $S \subset \mathbb{R}^2$ on which the function $f : \mathbb{R}^2 \rightarrow \mathbb{R}$ defined by

$$f(x_1, x_2) = \exp(x_1) + \exp(x_2) + x_1 x_2$$

is convex.

Input:

`condition:=convex(exp(x1)+exp(x2)+x1*x2, [x1,x2])`

Output:

$[e^{x_1}e^{x_2} - 1 \geq 0]$

Input:

```
lin(condition)
```

(See Section 6.24.4 p.326.)

Output:

$$[e^{x_1+x_2} - 1 \geq 0]$$

From here you conclude that f is convex when $x_1 + x_2 \geq 0$. The set S is therefore the half-space defined by this inequality.

The algorithm respects the assumptions that may be set upon variables. Therefore, the convexity of a given function can be checked on a particular domain.

Examples.

- *Input:*

```
assume(x1>0), assume(x2>0) ;;
convex(exp(x1)+exp(x2)+x1*x2, [x1,x2])
```

Output:

true

- *Input:*

```
assume(x>=0 and x<=pi/4) ;;
convex(exp(y)*sec(x)^3-z, [x,y,z])
```

Output:

true

6.22 Calculus of variations

6.22.1 The Brachistochrone Problem

The *Brachistochrone problem* is perhaps the original problem in the calculus of variations. The problem is to find the curve from two points in a plane such that an object falling under its own weight will get from the first point to the second in the shortest time.

If the points are $(0, y_0)$ and $(x_1, 0)$, with $y_0 > 0$ and $x_1 > 0$, this becomes the problem of minimizing the objective functional

$$T(y) = \int_0^{x_1} L(t, y(x), y'(x)) dx$$

where the function L is defined by

$$L(t, y(x), y'(x)) = \sqrt{\frac{1 + y'(x)^2}{2gy(x)}}$$

for $y : [0, x_1] \rightarrow \mathbb{R}$ such that $y(0) = y_0$ and $y(x_1) = 0$ (the constant g is the gravitational acceleration).

More generally, one type of problem in the Calculus of variations is to minimize (or maximize) a functional

$$F(y) = \int_a^b f(x, y, y') \, dx$$

over all functions $y \in C^2[a, b]$ with boundary conditions $y(a) = A$ and $y(b) = B$, where $A, B \in \mathbb{R}$. The function f is called the *Lagrangian*.

6.22.2 Euler-Lagrange equation(s): `euler_lagrange`

The *Euler-Lagrange equations* for a Lagrangian function $f(x, y, y')$ are differential equations which must be satisfied by extrema of the functional $F(y)$.

The `euler_lagrange` command finds the Euler-Lagrange equations for a Lagrangian f . The function f can be given in one of two ways. For the first way:

- `euler_lagrange` takes one mandatory argument and two optional arguments:
 - *expr*, an expression involving an independent variable, a dependent variable, and the dependent variable prime.
 - Optionally, *indvar*, the independent variable (by default x).
 - Optionally, *depvar*, the dependent variable (by default y).
 If a function $y \in \mathbb{R}^n$ is required (by default $n = 1$), you can enter $y = (y_1, y_2, \dots, y_n)$ as a vector $[y_1, y_2, \dots, y_n]$. In that case, $y' = (y'_1, y'_2, \dots, y'_n)$.

An alternate way to specify the independent and dependent variables is by replacing both optional arguments by either, for example, $y(x)$ or $[y_1(x), y_2(x), \dots, y_n(x)]$.

- `euler_lagrange(expr <, indvar, depvar >)` returns the system of differential Euler-Lagrange equations.

If $n = 1$, a single equation is returned:

$$\frac{\partial f}{\partial y} = \frac{d}{dx} \frac{\partial f}{\partial y'}. \quad (6.1)$$

In general, n equations are returned:

$$\frac{\partial f}{\partial y_k} = \frac{d}{dx} \frac{\partial f}{\partial y'_k}, \quad k = 1, 2, \dots, n.$$

The degrees of these differential equations are kept as low as possible. If, for example, $\frac{\partial f}{\partial y} = 0$, the equation $\frac{\partial f}{\partial y'} = K$ is returned, where $K \in \mathbb{R}$ is an arbitrary constant. Similarly, using the *Hamiltonian*

$$H(x, y, y') = y' \frac{\partial}{\partial y'} f(x, y, y') - f(x, y, y')$$

the Euler-Lagrange equation is simplified in the case $n = 1$ and $\frac{\partial f}{\partial t} = 0$ to:

$$H(x, y, y') = K, \quad (6.2)$$

since it can be shown that $\frac{d}{dx} H(y, y', x) = 0$. Therefore the Euler-Lagrange equations, which are generally of order two in y , are returned in a simpler form of order one in the aforementioned cases. If $n = 1$ and $\frac{\partial f}{\partial x} = 0$, then both equations (6.1) and (6.2) are returned, each of them being sufficient to determine y (one of the returned equations is usually simpler than the other).

Example.

Input:

```
euler_lagrange(sqrt(x'(t)^2+y'(t)^2),[x(t),y(t)])
```

Output

$$\left[\frac{\frac{d}{dt}x(t)}{\sqrt{\left(\frac{d}{dt}x(t)\right)^2 + \left(\frac{d}{dt}y(t)\right)^2}} = K_0, \frac{\frac{d}{dt}y(t)}{\sqrt{\left(\frac{d}{dt}x(t)\right)^2 + \left(\frac{d}{dt}y(t)\right)^2}} = K_1 \right]$$

where $K_0, K_1 \in \mathbb{R}$ are arbitrary (these constants are generated automatically).

It can be proven that if f is convex (as a function of three independent variables, see Section 6.21.8 p.296), then a solution y to the Euler-Lagrange equations minimizes the functional F .

Example.

Minimize the functional F for $0 < a < b$ and

$$f(x, y, y') = x^2 y'(x)^2 + y(x)^2.$$

Input:

```
eq:=euler_lagrange(x^2*diff(y(x),x)^2 + y^2)
```

Output:

$$\frac{d^2}{dx^2}y(x) = \frac{-2\frac{d}{dx}y(x)x + y(x)}{x^2}$$

This can be solved by assuming $y(x) = x^r$ for some $r \in \mathbb{R}$.

Input:

```
solve(subs(eq,y(x)=x^r),r)
```

Output:

$$\left[-\frac{\sqrt{5}+1}{2}, -\frac{-\sqrt{5}+1}{2} \right]$$

Note that a pair of independent solutions is also returned by the `kovacicsols` command (see Section 6.57.3 p.643):

Input:

```
assume(x>=0)::
kovacicsols(y'=(y-2x*y')/x^2,x,y)
```

Output:

$$\left[\sqrt{x^{\sqrt{5}-1}}, \sqrt{x^{-\sqrt{5}-1}} \right]$$

You can conclude that $y = C_1 x^{-\frac{\sqrt{5}+1}{2}} + C_2 x^{\frac{\sqrt{5}+1}{2}}$. The values of C_1 and C_2 are determined from the boundary conditions. Finally, to prove that f is convex:

Input:

```
convex(x^2*diff(y(x),x)^2 + y^2,y(x))
```

Output:

true

Therefore, y minimizes F on $[a, b]$.

Example.

Find the function y in

$$\left\{ y \in C^1 \left[\frac{1}{2}, 1 \right] : y \left(\frac{1}{2} \right) = -\frac{\sqrt{3}}{2}, y(1) = 0 \right\}$$

which minimizes the functional

$$F(y) = \int_{1/2}^1 \frac{\sqrt{1 + y'(x)^2}}{x} dx.$$

To obtain the corresponding Euler-Lagrange equation:

Input:

```
eq := euler_lagrange(sqrt(1+diff(y(x),x)^2)/x)
```

Output:

$$\frac{\frac{d}{dx}y(x)}{x\sqrt{\left(\frac{d}{dx}y(x)\right)^2 + 1}} = K_2$$

Input:

```
sol:=dsolve(eq)
```

Output:

$$\left[-\frac{\sqrt{-K_3^2 x^2 + 1}}{K_3} + c_0 \right]$$

The sought solution is the function of the above form which satisfies the boundary conditions.

Input:

```
y0:=sol[0]::
c:=[K_3,c_0]::
v:=solve([subs(y0,x=1/2)=-sqrt(3)/2,subs(y0,x=1)=0],c)
```

Output:

$$\begin{bmatrix} 1 & 0 \end{bmatrix}$$

Input:

$$y0:=\text{normal}(\text{subs}(y0,c,v[0]))$$

Output:

$$-\sqrt{-x^2+1}$$

To prove that $y_0(x) = -\sqrt{1-x^2}$ is indeed a minimizer for F , you need to show that the integrand in $F(y)$ is convex.

Input:

$$\text{convex}(\text{sqrt}(1+y'^2)/x, y(x))$$

Output:

$$[x \geq 0]$$

You can similarly find the minimizer for

$$F(y) = \int_0^\pi (2 \sin(x) y(x) + y'(x)^2) \, dx$$

where $y \in C^1[0, \pi]$ and $y(0) = y(\pi) = 0$.

Input:

$$\text{eq}:=\text{euler_lagrange}(2*\sin(x)*y(x)+\text{diff}(y(x),x)^2)$$

Output:

$$\frac{d^2}{dx^2} y(x) = \sin x$$

Input:

$$\text{dsolve}(\text{eq and } y(0)=0 \text{ and } y(\pi)=0, x, y)$$

Output:

$$-\sin x$$

The above function is the sought minimizer as the integrand $2 \sin(x) y(x) + y'(x)^2$ is convex:

Input:

$$\text{convex}(2*\sin(x)*y(x)+\text{diff}(y(x),x)^2, y(x))$$

Output:

$$\text{true}$$

Example.

Minimize the functional $F(y) = \int_0^1 (y'(x)^4 - 4y(x)) \, dx$ on $C^1[0, 1]$ with boundary conditions $y(0) = 1$ and $y(1) = 2$.

First, solve the associated Euler-Lagrange equation:

Input:

$$\text{eq}:=\text{euler_lagrange}(y'^4-4y, x, y)$$

Output:

$$\left[3 \left(\frac{d}{dx} y(x) \right)^4 + 4y(x) = K_6, \frac{d^2}{dx^2} y(x) = -\frac{1}{3 \left(\frac{d}{dx} y(x) \right)^2} \right]$$

Input:

```
dsolve(eq[1] and y(0)=1 and y(1)=2,x,y)
```

Output:

$$\left[-\frac{3}{4} (-x + 1.52832425067)^{\frac{4}{3}} + 2.32032831141 \right]$$

$$[-3*(-x+1.52832425067)^(4/3)/4+2.32032831141]$$

Next, check if the integrand in $F(y)$ is convex:

Input:

```
convex(y'^4-4y,[x,y])
```

Output:

true

Hence the minimizer is

$$y_0(x) = -\frac{3}{4} (1.52832425067 - x)^{4/3} + 2.32032831141, \quad 0 \leq x \leq 1.$$

6.22.3 Solution of the Brachistochrone Problem

To solve the brachistochrone problem (see Section 6.22.1 p.298), you can first find the Euler-Lagrange equations for the Lagrangian

$$L(x, y(x), y'(x)) = \sqrt{\frac{1 + y'(x)^2}{2gy(x)}}$$

You can simplify this somewhat by assuming that you are using units where $2g = 1$.

Input:

```
assume(y>=0):;
euler_lagrange(sqrt((1+y'^2)/y),x,y)
```

Output:

$$\left[-\frac{1}{\sqrt{\left(\left(\frac{d}{dx} y(x) \right)^2 + 1 \right) y(x)}} = K_2, \frac{d^2}{dx^2} y(x) = \frac{-\left(\frac{d}{dx} y(x) \right)^2 - 1}{2y(x)} \right]$$

It is easier to solve the first equation for y , since it is first-order and separable.

The first equation can be rewritten as

$$\frac{dy}{dx} = -\sqrt{\frac{C}{x} - 1}$$

for appropriate C , which can be solved by separation of variables, getting you the parametric equations

$$\begin{aligned}x &= \frac{1}{2}C(2\theta - \sin(2\theta)) \\ y &= \frac{1}{2}C(1 - \cos(2\theta))\end{aligned}$$

which parameterize a cycloid. This implicitly defines a function $y = \bar{y}(x)$ as the only stationary function for L . The problem is to prove that it minimizes T , which would be easy if the integrand L was convex. However, it's not the case here:

Input:

```
assume(y>=0)::
convex(sqrt((1+y'^2)/y),y(x))
```

Output:

$$\left[- \left(\frac{d}{dx} y(x) \right)^2 + 3 \geq 0 \right]$$

This is equivalent to $|y'(t)| \leq \sqrt{3}$, which is certainly not satisfied by the cycloid \bar{y} near the point $x = 0$.

Using the substitution $y(x) = z(x)^2/2$, you get $y'(x) = z'(x)z(x)$ and

$$L(x, y(x), y'(x)) = P(x, z(x), z'(x)) = \sqrt{2(z(x)^{-2} + z'(x)^2)}.$$

The function P is convex:

Input:

```
assume(z>=0)::
convex(sqrt(2(z^(-2)+z'^2)),z(x))
```

Output:

true

Hence the function $\bar{z}(t) = \sqrt{2\bar{y}(t)}$, stationary for P (which is verified directly), minimizes the objective functional

$$U(z) = \int_0^{x_1} P(x, z(x), z'(x)) dx.$$

From here and $U(z) = T(y)$ it easily follows that \bar{y} minimizes T and is therefore the brachistochrone. (For details see John L. Troutman, *Variational Calculus and Optimal Control* (second edition), page 257.)

6.22.4 Jacobi equation: jacobi_equation

To determine whether a solution y_0 to the Euler-Lagrange equations is an extrema, checking the convexity of the Lagrangian f doesn't always work. Another approach is to look at the Jacobi equation, which is

$$-\frac{d}{dt} (f_{y'y'}(y_0, y'_0, t) h') + \left(f_{yy}(y_0, y'_0, t) - \frac{d}{dt} f_{yy'}(y_0, y'_0, t) \right) h = 0. \quad (6.3)$$

for unknown function h . If the Jacobi equation has a solution such that $h(a) = 0$, $h(c) = 0$ for some $c \in (a, b]$ (the interval given in the variational problem) and h not identically zero on $[a, c]$, then c is called a *conjugate* to a . If a conjugate exists, then y_0 does not minimize the functional F . But the function y_0 minimizes F if $f_{y'y'}(y_0, y'_0, x) > 0$ for all $x \in [a, b]$ and there are no points conjugate to a in $(a, b]$.

The `jacobi_equation` command computes the Jacobi equation.

- `jacobi_equation` takes five or six arguments:
 - $f(y, y', x)$, an expression involving an independent variable, a dependent variable, and the dependent variable prime.
 - `depvar`, the independent variable.
 - `indvar`, the dependent variable. This argument and the previous one can be combined to a single argument `depvar(indvar)`, which case the call has five arguments.
 - expression y_0 representing a function in $C^1[a, b]$ which is stationary for the functional $F(y) = \int_a^b f(y, y', x) dx$.
 - h , a symbol for the unknown function in the Jacobi equation.
 - a , a real number which is the lower bound for x .
- `jacobi_equation($f(y, y', x), x, y, y_0, a$)` returns the Jacobi equation and possibly the solution.

If the Jacobi equation can be solved by `dsolve` (see Section 6.57.1 p.633), a sequence containing the equation (6.3) and its solution is returned. Otherwise, if (6.3) cannot be solved immediately, only the Jacobi equation is returned.

Example.

Input:

```
jacobi_equation(-1/2*y'(t)^2+y(t)^2/2,t,y,sin(t),h,0)
```

Output:

$$-\frac{d^2}{dt^2}h(t) - h(t) = 0, c_0 \sin t$$

6.22.5 Finding conjugate points: conjugate_equation

The `conjugate_equation` computes conjugate points.

- `conjugate_equation` takes four arguments:
 - y_0 , an expression which depends on an independent variable and two parameters. The expression y_0 is assumed to represent a stationary function for the problem of minimizing some functional $F(y) = \int_a^b f(x, y, y') dx$.
 - $[\alpha, \beta]$, a list of parameters which y_0 depends on.
 - $[A, B]$, a list of the values of parameters α and β , respectively.

- x , the independent variable.
- a , a real number equal to the lower or to the upper bound for x .

- `conjugate_equation`($y_0, [\alpha, \beta], [A, B], x, a$) returns the expression

$$\frac{\partial y_0(t)}{\partial \alpha} \frac{\partial y_0(a)}{\partial \beta} - \frac{\partial y_0(a)}{\partial \alpha} \frac{\partial y_0(t)}{\partial \beta}, \quad (6.4)$$

at $\alpha = A$ and $\beta = B$, which is zero if and only if t is conjugate to a .

To find any conjugate points, set the returned expression to zero and solve.

Example.

Find a minimum for the functional

$$F(y) = \int_0^{\pi/2} (y'(x)^2 - x y(x) - y(x)^2) dx$$

on $D = \{y \in C^1[0, \pi/2] : y(0) = y(\pi/2) = 0\}$.

The corresponding Euler-Lagrange equation is:

Input:

```
eq:=euler_lagrange(y'(x)^2-x*y(x)-y(x)^2,y(x))
```

Output:

$$\frac{d^2}{dx^2} y(x) = -\frac{x}{2} - y(x)$$

The general solution is:

Input:

```
y0:=dsolve(eq,x,y)
```

Output:

$$c_0 \cos x + c_1 \sin x - \frac{x}{2}$$

The stationary function depends on two parameters c_0 and c_1 which are fixed by the boundary conditions:

Input:

```
c:=solve([subs(y0,x,0)=0,subs(y0,x,pi/2)=0],[c_0,c_1])
```

Output:

$$\left[\left[0, \frac{1}{4}\pi \right] \right]$$

Input:

```
conjugate_equation(y0,[c_0,c_1],c[0],x,0)
```

Output:

$$\sin x$$

The above expression obviously has no zeros in $(0, \pi/2]$, hence there are no points conjugate to 0. Since $f_{y'y'} = 2 > 0$, where $f(y, y', x)$ is the integrand in $F(y)$ (the strong Legendre condition), y_0 minimizes F on D . To obtain y_0 explicitly:

Input:

```
subs(y0, [c_0, c_1], c[0])
```

Output:

$$\frac{1}{4}\pi \sin x - \frac{x}{2}$$

6.22.6 An example: Finding the surface of revolution with minimal area

In this section, you will find the function

$$y_0 \in D = \{y \in C^1[0, 1] : y(0) = 1, y(1) = 2/3\}$$

for which the area of the corresponding surface of revolution is minimal. The result is not necessarily intuitive.

The area of the surface of revolution is measured by the functional

$$F(y) = 2\pi \int_0^1 y(x) \sqrt{1 + y'(x)^2} dx.$$

First, set $f(y, y', x) = y(x) \sqrt{1 + y'(x)^2}$ and compute the associated Euler-Lagrange equation:

Input:

```
eq := euler_lagrange(y(x)*sqrt(1+diff(y(x), x)^2))
```

Output:

$$\left[-\frac{y(x)}{\sqrt{\left(\frac{d}{dx}y(x)\right)^2 + 1}} = K_0, \frac{d^2}{dx^2}y(x) = \frac{\left(\frac{d}{dx}y(x)\right)^2 + 1}{y(x)} \right]$$

You can obtain the stationary function by finding the general solution of the first equation.

Input:

```
sol:=collect(simplify(dsolve(eq[0], x, y)))
```

(See Section 6.27.16 p.361). *Output:*

$$\left[-K_0, \frac{K_0 \left(-\left(e^{\frac{x-c_1}{K_0}} \right)^2 - 1 \right)}{2e^{\frac{x-c_1}{K_0}}} \right]$$

Obviously the constant solution $-K_0$ is not in D , so set y_0 to be the second element of the above list. That function, which can be written as

$$y_0(x) = -K_0 \cosh \left(\frac{x - c_1}{K_0} \right),$$

is a *catenary*.

Input:

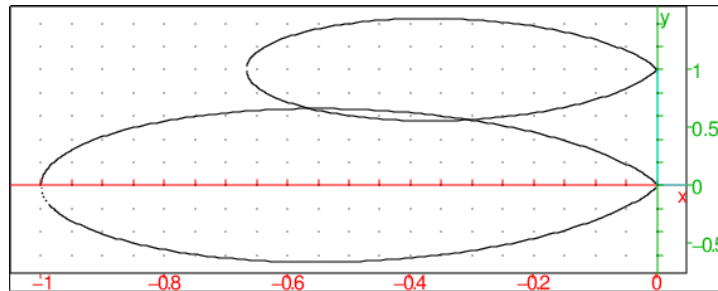
```
y0:=sol[1]:: p:=[K_0,c_1]::
```

To find the values of K_0 and c_1 from the boundary conditions, first plot the curves $y_0(0) = 1$ and $y_0(1) = \frac{2}{3}$ for $K_0 \in [-1, 1]$ and $c_1 \in [-1, 2]$ to see where they intersect each other.

Input:

```
eq1:=subs(y0,x=0)=1:: eq2:=subs(y0,x=1)=2/3::
implicitplot([eq1,eq2],K_0=-1..1,c_1=-1..2)
```

Output:



Observe that there are exactly two catenaries satisfying the Euler-Lagrange necessary conditions and the given boundary conditions: the first with $K_0 \approx -0.5$ and $c_1 \approx 0.6$ and the second with $K_0 \approx -0.3$ and $c_1 \approx 0.5$. You can obtain the values of these constants more precisely by using `fsolve`.

Input:

```
p1:=fsolve([eq1,eq2],p,[-0.5,0.6]);
p2:=fsolve([eq1,eq2],p,[-0.3,0.5])
```

Output:

```
[-0.56237423894, 0.662588703113], [-0.30613431407, 0.567138261119]
```

You can check, for each catenary, whether the strong Legendre condition

$$f_{y'y'}(x, y_k, y'_k) > 0$$

holds for $k = 1, 2$.

Input:

```
y1:=subs(y0,p,p1):: y2:=subs(y0,p,p2)::
D2f:=diff(f,diff(y(x),x),2)::
solve([eval(subs(D2f,y=y1,y(x)=y1))<=0,x>=0,x<=1],x);
solve([eval(subs(D2f,y=y2,y(x)=y2))<=0,x>=0,x<=1],x)
```

Output:

```
[], []
```

You can conclude that the strong Legendre condition is satisfied in both cases, so you can proceed by attempting to find the points conjugate to 0 for each catenary. The function y_0 depends on two parameters, so you can use `conjugate_equation` to find these points easily.

Input:

```
fsolve(conjugate_equation(y0,p,p1,x,0)=0,x=0..1)
fsolve(conjugate_equation(y0,p,p2,x,0)=0,x=0..1)
```

Output:

```
[0.0], [0.0, 0.799514772606]
```

You can conclude that there are no points conjugate to 0 in $(0, 1]$ for the catenary y_1 , so it minimizes the functional F . However, for the other catenary there is a conjugate point in the relevant interval, therefore y_2 is not a minimizer.

You can verify the above conclusions by computing the surface area for catenaries y_1 and y_2 and comparing them.

Input:

```
int(y1*sqrt(1+diff(y1,x)^2),x=0..1);
int(y2*sqrt(1+diff(y2,x)^2),x=0..1)
```

Output:

```
0.81396915825, 0.826468466845
```

You can see that the surface formed by rotating the curve y_1 is indeed smaller than the area of the surface formed by rotating the curve y_2 . Finally, you can visualize both surfaces for convenience.

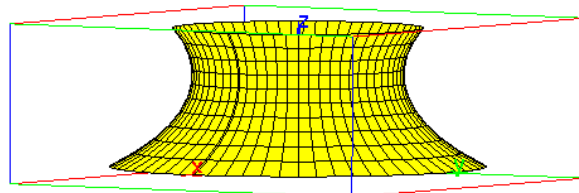
Input:

(see Section 8.6 p.674 for information on plot3d)

```
plot3d([y1*cos(t),y1*sin(t),x],x=0..1,t=0..2*pi,
        display=yellow+filled)
```

Output:

```
mouse plan 0.797x+0.544y+0.261z=0.124
```

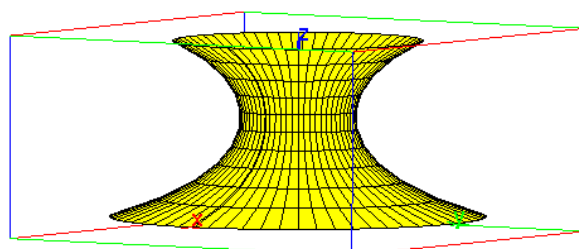


Input:

```
plot3d([y2*cos(t),y2*sin(t),x],x=0..1,t=0..2*pi,
        display=yellow+filled)
```

Output:

```
mouse plan 0.797x+0.544y+0.261z=0.124
```



6.23 Trigonometry

Xcas can evaluate the trigonometric functions in either radians or degrees (see Section 6.16.2 p.246). It can also manipulate them algebraically.

6.23.1 Expanding a trigonometric expression: `trigexpand`

The `trigexpand` command expands sums, differences and products by an integer inside the trigonometric functions.

- `trigexpand` takes one argument:
expr, an expression containing trigonometric functions.
- `trigexpand(expr)` returns the expression with sums, differences and integer products inside the trigonometric functions expanded.

Input:

```
trigexpand(cos(x+y))
```

Output:

$$\cos x \cdot \cos y - \sin x \cdot \sin y$$

6.23.2 Linearizing a trigonometric expression: `tlin`

The `tlin` command linearizes products and integer powers of the trigonometric functions (e.g. in terms of $\sin(n \cdot x)$ and $\cos(n \cdot x)$).

- `tlin` takes one argument:
expr, an expression containing trigonometric functions.
- `tlin(expr)` returns the expression with the trigonometric functions linearized.

Examples.

- Linearize $\cos(x) \cdot \cos(y)$.

Input:

```
tlin(cos(x)*cos(y))
```

Output:

$$\frac{\cos(x-y)}{2} + \frac{\cos(x+y)}{2}$$

- Linearize $\cos(x)^3$.

Input:

```
tlin(cos(x)^3)
```

Output:

$$\frac{3}{4} \cos x + \frac{\cos(3x)}{4}$$

- Linearize $4 \cos(x)^2 - 2$.

Input:

```
tlin(4*cos(x)^2-2)
```

Output:

$$2 \cos(2x)$$

6.23.3 Increasing the phase by $\pi/2$ in a trigonometric expression: `shift_phase`

The `shift_phase` command increases the phase of a trigonometric expression by $\pi/2$.

- `shift_phase` takes one argument:
expr, a trigonometric expression.
- `shift_phase(expr)` returns *expr* with the phase increased by $\pi/2$ (after automatic simplification).

Examples.

- *Input:*

```
shift_phase(x + sin(x))
```

Output:

$$x - \cos\left(\frac{\pi + 2x}{2}\right)$$

- *Input:*

```
shift_phase(x + cos(x))
```

Output:

$$x + \sin\left(\frac{\pi + 2x}{2}\right)$$

- *Input:*

```
shift_phase(x + tan(x))
```

Output:

$$x - \frac{1}{\tan\left(\frac{\pi+2x}{2}\right)}$$

Quoting the argument will prevent the automatic simplification.

Example.

Input:

```
shift_phase('sin(x + pi/2)')
```

Output:

$$-\cos\left(\frac{\pi + 2x + 2\frac{\pi}{2}}{2}\right)$$

With an unquoted sine, you get:

Input:

```
shift_phase(sin(x + pi/2))
```

Output:

$$\sin\left(\frac{\pi + 2x}{2}\right)$$

since `sin(x+pi/2)` is evaluated (in this case simplified) before `shift_phase` is called, and `shift_phase(cos(x))` returns `sin((pi+2*x)/2)`.

6.23.4 Putting together sine and cosine of the same angle: `tcollect` `tCollect`

The `tcollect` command linearizes trigonometric expressions (in terms of $\sin(n * x)$ and $\cos(n * x)$) and combines sines and cosines of the same angle. `tCollect` is a synonym for `tcollect`.

- `tcollect` takes one argument:
expr, an expression containing trigonometric functions.
- `tcollect(expr)` returns *expr* after first linearizing it and then combining sines and cosines of the same angle.

Examples.

- *Input:*

```
tcollect(sin(x)+cos(x))
```

Output:

$$\sqrt{2} \cos\left(x - \frac{1}{4}\pi\right)$$

- *Input:*

```
tcollect(2*sin(x)*cos(x)+cos(2*x))
```

Output:

$$\sqrt{2} \cos\left(2x - \frac{1}{4}\pi\right)$$

6.23.5 Simplifying: `simplify`

The `simplify` command simplifies expressions. As with all automatic simplifications, do not expect miracles; you will have to use specific rewriting rules if it does not work.

- `simplify` takes one argument:
expr, an expression.
- `simplify(expr)` returns the simplified version of *expr*.

Example.

Input:

```
simplify((sin(3*x)+sin(7*x))/sin(5*x))
```

Output:

$$2 \cos(2x)$$

Warning.

`simplify` is more efficient in `radian` mode (which you can turn on, if it isn't already, by checking `radian` in the `cas` configuration or inputting `angle_radian:=1`, see Section 3.5.3 p.71).

6.23.6 Simplifying trigonometric expressions: `trigsimplify`

The `trigsimplify` command simplifies trigonometric expressions by combining `simplify` (see Section 6.12.14 p.213), `texpand` (see Section 6.25.1 p.329), `tlin` (see Section 6.23.2 p.310), `tcollect` (see Section 6.23.4 p.312), `trigsin` (see Section 6.23.23 p.321), `trigcos` (see Section 6.23.24 p.322) and `trigtan` (see Section 6.23.25 p.322) commands in a certain order.

- `trigsimplify` takes one argument:
expr, an argument containing trigonometric functions.
- `trigsimplify(expr)` returns the simplified form of *expr*.

Examples.

- *Input:*

```
trigsimplify((sin(x+y)-sin(x-y))/(cos(x+y)+cos(x-y)))
```

Output:

$$\tan y$$

- *Input:*

```
trigsimplify(1-1/4*sin(2a)^2-sin(b)^2-cos(a)^4)
```

Output:

$$\sin^2 a - \sin^2 b$$

6.23.7 Transforming arccos into arcsin: acos2asin

The `acos2asin` command transforms any `acoss` in an expression to `asins`, using the identity $\arccos(x) = \pi/2 - \arcsin(x)$.

- `acos2asin` takes one argument:
`expr`, an expression containing inverse trigonometric functions.
- `acos2asin(expr)` returns `expr` with any `acoss` replaced by the appropriate `asins`.

Example.

Input:

`acos2asin(acos(x)+asin(x))`

Output (after simplification):

$$\frac{\pi}{2}$$

6.23.8 Transforming arccos into arctan: acos2atan

The `acos2atan` command transforms any `acoss` in an expression to `atans`, using the identity

$$\arccos(x) = \frac{\pi}{2} - \arctan\left(\frac{x}{\sqrt{1-x^2}}\right)$$

- `acos2atan` takes one argument:
`expr`, an expression containing inverse trigonometric functions.
- `acos2atan(expr)` returns `expr` with any `acoss` replaced by the appropriate `atans`.

Example.

Input:

`acos2atan(acos(x))`

Output:

$$\frac{\pi}{2} - \arctan\left(\frac{x}{\sqrt{1-x^2}}\right)$$

6.23.9 Transforming arcsin into arccos: asin2acos

The `asin2acos` command transforms any `asins` in an expression to `acoss`, using the identity $\arcsin(x) = \pi/2 - \arccos(x)$.

- `asin2acos` takes one argument:
`expr`, an expression containing inverse trigonometric functions.
- `asin2acos(expr)` returns `expr` with any `asins` replaced by the appropriate `acoss`.

Example.*Input:*`asin2acos(cos(x)+sin(x))`*Output (after simplification):*

$$\frac{\pi}{2}$$

6.23.10 Transforming arcsin into arctan: asin2atan

The `asin2atan` command transforms any `asins` in an expression to `atans`, using the identity

$$\arcsin(x) = \arctan\left(\frac{x}{\sqrt{1-x^2}}\right)$$

- `asin2atan` takes one argument:
`expr`, an expression containing inverse trigonometric functions.
- `asin2atan(expr)` returns `expr` with any `asins` replaced by the appropriate `atans`.

Example.*Input:*`asin2atan(sin(x))`*Output:*

$$\arctan\left(\frac{x}{\sqrt{1-x^2}}\right)$$

6.23.11 Transforming arctan into arcsin: atan2asin

The `atan2asin` command transforms any `atans` in an expression to `asins`, using the identity

$$\arctan(x) = \arcsin\left(\frac{x}{\sqrt{1+x^2}}\right)$$

- `atan2asin` takes one argument:
`expr`, an expression containing inverse trigonometric functions.
- `atan2asin(expr)` returns `expr` with any `atans` replaced by the appropriate `asins`.

Example.*Input:*`atan2asin(atan(x))`*Output:*

$$\arcsin\left(\frac{x}{\sqrt{1+x^2}}\right)$$

6.23.12 Transforming arctan into arccos: atan2acos

The `atan2acos` command transforms any `atans` in an expression to `acoss`, using the identity

$$\arctan(x) = \frac{\pi}{2} - \arcsin\left(\frac{x}{\sqrt{1+x^2}}\right)$$

- `atan2acos` takes one argument:
`expr`, an expression containing inverse trigonometric functions.
- `atan2acos(expr)` returns `expr` with any `atans` replaced by the appropriate `acoss`.

Example.

Input:

`atan2acos(atan(x))`

Output:

$$\frac{\pi}{2} - \arccos\left(\frac{x}{\sqrt{1+x^2}}\right)$$

6.23.13 Transforming complex exponentials into sin and cos: sincos exp2trig

The `sincos` command uses the identity

$$e^{ix} = \cos(x) + i \sin(x)$$

to rewrite complex exponentials in terms of sine and cosine. `exp2trig` is a synonym for `sincos`.

- `sincos` takes one argument:
`expr`, an expression containing complex exponentials.
- `sincos(expr)` rewrites `expr` in terms of sin and cos.

Examples.

- *Input:*

`sincos(exp(i*x))`

Output:

$$\cos x + i \sin x$$

- *Input:*

`exp2trig(exp(-i*x))`

Output:

$$\cos x - i \sin x$$

- *Input:*

```
simplify(sincos(((i)*(exp((i)*x))^2-i)/(2*exp((i)*x))))
```

or:

```
simplify(exp2trig(((i)*(exp((i)*x))^2-i)/(2*exp((i)*x))))
```

Output:

$$-\sin x$$

6.23.14 Transforming $\tan(x)$ into $\sin(x)/\cos(x)$: `tan2sincos`

The `tan2sincos` command replaces $\tan(x)$ by $\frac{\sin(x)}{\cos(x)}$ in an expression.

- `tan2sincos` takes one argument:
expr, an expression containing trigonometric functions.
- `tan2sincos(expr)` returns *expr* with anything of the form $\tan(x)$ replaced by $\frac{\sin(x)}{\cos(x)}$.

Example.

Input:

```
tan2sincos(tan(2*x))
```

Output:

$$\frac{\sin(2x)}{\cos(2x)}$$

6.23.15 Transforming $\sin(x)$ into $\cos(x)*\tan(x)$: `sin2costan`

The `sin2costan` command replaces $\sin(x)$ by $\cos(x)\tan(x)$ in an expression.

- `sin2costan` takes one argument:
expr, an expression containing trigonometric functions.
- `sin2costan(expr)` returns *expr* with anything of the form $\sin(x)$ replaced by $\cos(x)\tan(x)$.

Example.

Input:

```
sin2costan(sin(2*x))
```

Output:

$$\tan(2x)\cos(2x)$$

6.23.16 Transforming $\cos(x)$ into $\sin(x)/\tan(x)$: `cos2sintan`

The `cos2sintan` command replaces $\cos(x)$ by $\frac{\sin(x)}{\tan(x)}$ in an expression.

- `cos2sintan` takes one argument:
expr, an expression containing trigonometric functions.
- `cos2sintan(expr)` returns *expr* with anything of the form $\cos(x)$ replaced by $\frac{\sin(x)}{\tan(x)}$.

Example.

Input:

`cos2sintan(cos(2*x))`

Output:

$$\frac{\sin(2x)}{\tan(2x)}$$

6.23.17 Rewriting $\tan(x)$ in terms of $\sin(2x)$ and $\cos(2x)$: `tan2sincos2`

The `tan2sincos2` command replaces $\tan(x)$ by $\frac{\sin(2x)}{1 + \cos(2x)}$ in an expression.

- `tan2sincos2` takes one argument:
expr, an expression containing trigonometric functions.
- `tan2sincos2(expr)` returns *expr* with anything of the form $\tan(x)$ replaced by $\frac{\sin(2x)}{1 + \cos(2x)}$.

Example.

Input:

`tan2sincos2(tan(x))`

Output:

$$\frac{\sin(2x)}{1 + \cos(2x)}$$

6.23.18 Rewriting $\tan(x)$ in terms of $\cos(2x)$ and $\sin(2x)$: `tan2cossin2`

The `tan2cossin2` command replaces $\tan(x)$ by $\frac{1 - \cos(2x)}{\sin(2x)}$ in an expression.

- `tan2cossin2` takes one argument:
expr, an expression containing trigonometric functions.
- `tan2cossin2(expr)` returns *expr* with anything of the form $\tan(x)$ replaced by $\frac{1 - \cos(2x)}{\sin(2x)}$.

Example.*Input:*`tan2cossin2(tan(x))`*Output:*

$$\frac{1 - \cos(2x)}{\sin(2x)}$$

6.23.19 Rewriting sin, cos, tan in terms of tan(x/2): halftan

The `halftan` command rewrites the trigonometric functions in terms of $\tan(x/2)$ using the identities:

$$\begin{aligned}\sin(x) &= \frac{2 \tan\left(\frac{x}{2}\right)}{\tan^2\left(\frac{x}{2}\right) + 1} \\ \cos(x) &= \frac{1 - \tan^2\left(\frac{x}{2}\right)}{\tan^2\left(\frac{x}{2}\right) + 1} \\ \tan(x) &= \frac{2 \tan\left(\frac{x}{2}\right)}{1 - \tan^2\left(\frac{x}{2}\right)}\end{aligned}$$

- `halftan` takes one argument:
expr, an expression containing trigonometric functions.
- `halftan(expr)` returns *expr* with any trigonometric functions replaced by the appropriate expression of $\tan(x/2)$.

Examples.

- *Input:*

`halftan(sin(2*x)/(1+cos(2*x)))`*Output:*

$$\frac{2 \tan\left(\frac{2}{2}x\right)}{(\tan^2\left(\frac{2}{2}x\right) + 1) \left(1 + \frac{1 - \tan^2\left(\frac{2}{2}x\right)}{\tan^2\left(\frac{2}{2}x\right) + 1}\right)}$$

Output (after simplification with normal(ans())):

$$\tan x$$

- *Input:*

`halftan(sin(x)^2+cos(x)^2)`*Output:*

$$\left(\frac{2 \tan\left(\frac{x}{2}\right)}{\tan^2\left(\frac{x}{2}\right) + 1}\right)^2 + \left(\frac{1 - \tan^2\left(\frac{x}{2}\right)}{\tan^2\left(\frac{x}{2}\right) + 1}\right)^2$$

Output (after simplification with normal(ans())):

6.23.20 Rewriting trigonometric functions in terms of $\tan(x/2)$ and hyperbolic functions in terms of $\exp(x)$: `halftan_hyp2exp`

The `halftan_hyp2exp` command rewrites the trigonometric function in terms of $\tan(x/2)$ (like `halftan`, see Section 6.23.19 p.319) and rewrites the hyperbolic functions in terms of their definitions using exponentials, namely:

$$\begin{aligned}\sinh(x) &= \frac{e^x - e^{-x}}{2} \\ \cosh(x) &= \frac{e^x + e^{-x}}{2} \\ \tanh(x) &= \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1}\end{aligned}$$

- `halftan_hyp2exp` takes one argument: *expr*, a trigonometric and hyperbolic expression.
- `halftan_hyp2exp(expr)` returns *expr* with any trigonometric functions replaced by the appropriate expression in $\tan(x/2)$ and any hyperbolic functions replaced by the appropriate exponentials.

Examples.

- *Input:*

`halftan_hyp2exp(tan(x)+tanh(x))`

Output:

$$\frac{2 \tan\left(\frac{x}{2}\right)}{1 - \tan^2\left(\frac{x}{2}\right)} + \frac{e^{2x} - 1}{e^{2x} + 1}$$

- *Input:*

`halftan_hyp2exp(sin(x)^2+cos(x)^2-sinh(x)^2+cosh(x)^2)`

Output (after simplification with `normal(ans())`):

2

6.23.21 Transforming trigonometric functions into complex exponentials : `trig2exp`

The `trig2exp` command replaces trigonometric functions by their complex exponential form.

- `trig2exp` takes one argument: *expr*, an expression containing trigonometric functions.
- `trig2exp(expr)` returns *expr* with the trigonometric functions replaced by the appropriate complex exponentials (WITHOUT linearization).

Examples.

- *Input:*

$$\text{trig2exp}(\tan(x))$$

Output:

$$\frac{(e^{ix})^2 - 1}{i((e^{ix})^2 + 1)}$$

- *Input:*

$$\text{trig2exp}(\sin(x))$$

Output:

$$\frac{e^{ix} - \frac{1}{e^{ix}}}{2i}$$

6.23.22 Transforming inverse trigonometric functions into logarithms: `atrig2ln`

Just as the trigonometric functions can be written in terms of complex exponentials, the inverse trigonometric functions can be written in terms of complex logarithms. The `atrig2ln` command does this rewriting.

- `atrig2ln` takes one argument: *expr*, an expression containing inverse trigonometric functions.
- `atrig2ln(expr)` returns *expr* with any inverse trigonometric functions replaced by the appropriate complex logarithms.

Example.

Input:

$$\text{atrig2ln}(\text{asin}(x))$$

Output:

$$i \ln \left(x + \sqrt{x^2 - 1} \right) + \frac{\pi}{2}$$

6.23.23 Simplifying and expressing preferentially with sines: `trigsin`

Any trigonometric function can be written in terms of sines and cosines, and with the identity $\sin^2(x) + \cos^2(x) = 1$, the even powers of \cos can be turned into powers of \sin . The `trigsin` command performs these substitutions.

- `trigsin` takes one argument:
expr, an expression containing trigonometric functions.
- `trigsin(expr)` returns *expr* with the trigonometric functions rewritten in terms of \sin and \cos , with as many \cos as possible transformed to \sin .

Example.*Input:*

$$\text{trigsin}(\sin(x)^4 + \cos(x)^2 + 1)$$
Output:

$$\sin^4 x - \sin^2 x + 2$$

6.23.24 Simplifying and expressing preferentially with cosines: trigcos

Any trigonometric function can be written in terms of sines and cosines, and with the identity $\sin(x)^2 + \cos(x)^2 = 1$, the even powers of sin can be turned into powers of cos. The **trigcos** command performs these substitutions.

- **trigcos** takes one argument:
expr, an expression containing trigonometric functions.
- **trigsin(*expr*)** returns *expr* with the trigonometric functions rewritten in terms of sin and cos, with as many sins as possible transformed to coss.

Example.*Input:*

$$\text{trigcos}(\sin(x)^4 + \cos(x)^2 + 1)$$
Output:

$$\cos^4 x - \cos^2 x + 2$$

6.23.25 Simplifying and expressing preferentially with tangents: trigtan

The **trigtan** command rewrites trigonometric expressions into expressions where as many trigonometric functions as possible are written in terms of tangents, using the identities $\sin(x)^2 + \cos(x)^2 = 1$, $\tan(x) = \frac{\sin(x)}{\cos(x)}$.

- **trigtan** takes one argument:
expr, an expression containing trigonometric functions.
- **trigtan(*expr*)** returns *expr* with the trigonometric functions written as much as possible in terms of tangents.

Example.*Input:*

$$\text{trigtan}(\sin(x)^4 + \cos(x)^2 + 1)$$
Output:

$$\frac{2 \tan^4 x + 3 \tan^2 x + 2}{\tan^4 x + 2 \tan^2 x + 1}$$

6.23.26 Rewriting an expression with different options: `convert` `convertir =>`

`Xcas` has many commands to convert expressions into different forms; the `convert` command (or its infix version `=>`) is a different way to call many of these functions.

- `convert` takes two or more arguments:
 - *expr*, an expression.
 - *option*, an option specifying which rewrite rules to use. A third argument might be necessary for some options. Possible values of *option* are:
 - * `sin`, to convert an expression like `trigsin` (see Section 6.23.23 p.321).
 - * `cos`, to convert an expression like `trigcos` (see Section 6.23.24 p.322).
 - * `sincos`, to convert an expression like `sincos` (see Section 6.23.13 p.316).
 - * `trig`, to convert an expression like `sincos` (see Section 6.23.13 p.316).
 - * `tan`, to convert an expression like `halftan` (see Section 6.23.19 p.319).
 - * `exp`, to convert an expression like `trig2exp` (see Section 6.23.21 p.320).
 - * `ln`, to convert an expression like `trig2exp` (see Section 6.23.21 p.320).
 - * `expln`, to convert an expression like `trig2exp` (see Section 6.23.21 p.320).
 - * `string`, to convert a expression into a string.
 - * `matrix`, to convert a list of lists into a matrix.
 - * `array`, to turn a table into an array (see Section 6.46.1 p.539).
 - * `polynom`, to convert a series (see Section 6.36.2 p.439) into a polynomial by removing the remainder (see Section 6.27.25 p.367) or to convert a list representing a polynomial into a polynomial in internal sparse multivariate form (see Section 6.27.2 p.351 and Section 6.27.6 p.354).
 - * `parfrac` (or `partfrac` or `fullparfrac`), to convert a rational function into its partial fraction decomposition (see Section 6.32.9 p.416).
 - * `interval`, to convert an expression which evaluates to a number into an interval (see Section 6.38.9 p.452).
 - * `list` (or no argument), to convert a polynomial in internal sparse multivariate format (see Section 6.27.2 p.351) into a list.
 - * `unit`, a unit, to convert a unit object to a new compatible unit (see Section 11.1.4 p.830).

The values of *option* that require a third argument:

- * **confrac**, to convert a number into a continued fraction. (See Section 6.7.7 p.166.) The third argument will be the name of a variable to store the continued fraction into (which must be quoted the variable was assigned).
- * **base**, to convert a number into a different base (beginning with the units digit). If **expr** is a number, then the third argument will be base to convert to (see Section 6.4.2 p.132), if *expr* is a list of numbers, then the third argument will be the base to convert from (and *expr* will be a list of the digits in this base, starting with the units digit).

Finally, if *expr* is an expression with units (see Section 11.1.1 p.827), then *option* can be new units to convert to (see Section 11.1.4 p.830).

- **convert**(*expr*,*option*[,*extraop*]) returns the expression with the requested conversions done.

Examples.

- *Input:*

```
convert(1.2,confrac,'fc')
```

Output:

```
[1, 5]
```

and **fc** contains the continued fraction equal to 1.2.

- *Input:*

```
convert(123,base,10)
```

Output:

```
[3, 2, 1]
```

- *Input:*

```
convert([3,2,1],base,10)
```

Output:

```
123
```

- *Input:*

```
convert(1000_g,_kg)
```

Output:

```
1.0 kg
```

6.24 Exponentials and Logarithms

6.24.1 Rewriting hyperbolic functions as exponentials: `hyp2exp`

The hyperbolic functions are typically defined in terms of exponential functions; the `hyp2exp` command converts hyperbolic functions into their exponential forms.

- `hyp2exp` takes one argument:
expr, an expression.
- `hyp2exp(expr)` rewrites each hyperbolic function in *expr* with exponentials (as a rational function of one exponential, i.e. WITHOUT linearization).

Example.

Input:

```
hyp2exp(sinh(x))
```

Output:

$$\frac{e^x - \frac{1}{e^x}}{2}$$

6.24.2 Expanding exponentials: `expexpand`

The exponential function applied to a sum can be converted into a product of exponentials; namely,

$$e^{x+y} = e^x e^y$$

The `expexpand` command does this conversion. (For expansions with other bases, see Section [6.24.6](#) p.327.)

- `expexpand` takes one argument:
expr, an expression.
- `expexpand(expr)` returns the expression *expr* with exponentials (base *e*) of sums rewritten as products of exponentials.

Example.

Input:

```
expexpand(exp(3*x)+exp(2*x+2))
```

Output:

$$(e^x)^3 + (e^x)^2 e^2$$

6.24.3 Expanding logarithms: `lnexpand`

The logarithm applied to a product can be converted into a sum of logarithms; namely,

$$\log(x \cdot y) = \log(x) + \log(y)$$

The `lnexpand` command does this expansion.

- `lnexpand` takes one argument:
expr, an expression.
- `lnexpand(expr)` returns the expression *expr* with logarithms of products rewritten as sums of logarithms.

Example.

Input:

```
lnexpand(ln(3*x^2)+ln(2*x+2))
```

Output:

$$\ln(3) + 2\ln|x| + \ln(2) + \ln(x + 1)$$

6.24.4 Linearizing exponentials: `lin`

The `lin` command will linearize expressions involving exponentials; namely, it will replace products of exponentials by exponentials of sums. It will first replace any hyperbolic functions by exponentials.

- `lin` takes one argument:
expr, an expression.
- `lin(expr)` returns the linearized version of *expr*.

Examples.

- *Input:*

```
lin(sinh(x)^2)
```

Output:

$$\frac{e^{2x}}{4} - \frac{1}{2} + \frac{e^{-2x}}{4}$$

- *Input:*

```
lin((exp(x)+1)^3)
```

Output:

$$e^{3x} + 3e^{2x} + 3e^x + 1$$

6.24.5 Collecting logarithms: `lncollect`

The `lncollect` command collects the logarithm in an expression; namely, it rewrites sums of logarithms as the logarithm of a product.

- `lncollect` takes one argument:
expr, an expression.
- `lncollect(expr)` returns *expr* with the logarithms collected.

It may be a good idea to factor the expression with `factor` before collecting by `lncollect`).

Examples.

- *Input:*

```
lncollect(ln(x+1)+ln(x-1))
```

Output:

$$\ln((x+1)(x-1))$$

- *Input:*

```
lncollect(exp(ln(x+1)+ln(x-1)))
```

Output:

$$(x+1)(x-1)$$

Warning!!!

For Xcas, `log` is the natural logarithm, the same as `ln`; for the base 10 logarithm, use `log10`.

6.24.6 Expanding powers: `powexpand`

The `powexpand` command rewrites a power of a sum as a product of powers; it is `expexpand` (see Section 6.24.2 p.325) with bases other than e.

- `powexpand` takes one argument:
expr, an expression.
- `powexpand(expr)` returns *expr* with powers of sums replaced by sums of powers.

Example.

Input:

```
powexpand(a^(x+y))
```

Output:

$$a^x a^y$$

6.24.7 Rewriting a power as an exponential: pow2exp

Powers with arbitrary (positive) bases are often defined in terms of exponentials with base e with

$$a^x = e^{x \ln(a)}$$

The `pow2exp` rewrites powers to exponentials.

- `pow2exp` takes one argument:
expr, an exponential.
- `pow2exp(expr)` returns *expr* with any powers replaced by their corresponding exponential.

Example.

Input:

```
pow2exp(a^(x+y))
```

Output:

$$e^{(x+y) \ln a}$$

6.24.8 Rewriting $\exp(n \cdot \ln(x))$ as a power: exp2pow

The `exp2pow` command is the inverse of `pow2exp` (see Section 6.24.7 p.328).

- `exp2pow` takes one argument:
expr, an expression.
- `exp2pow(expr)` rewrites any subexpressions of *expr* of the form $\exp(n \cdot \ln(x))$ as x^n .

Example.

Input:

```
exp2pow(exp(n*ln(x)))
```

Output:

$$x^n$$

Note the difference with `lncollect`:

```
lncollect(exp(n*ln(x))) = exp(n*log(x))
lncollect(exp(2*ln(x))) = exp(2*log(x))
exp2pow(exp(2*ln(x))) = x^2
```

but

```
lncollect(exp(ln(x)+ln(x))) = x^2
exp2pow(exp(ln(x)+ln(x))) = x^(1+1)
```


6.24.9 Simplifying complex exponentials: `tsimplify`

The `tsimplify` command simplifies transcendental expressions by rewriting the expression with complex exponentials. It is a good idea to try other simplification instructions and call `tsimplify` if they do not work.

- `tsimplify` takes one argument:
expr, an expression.
- `tsimplify(expr)` returns a (possibly) simplified version of *expr*.

Example.

Input:

```
tsimplify((sin(7*x)+sin(3*x))/sin(5*x))
```

Output:

$$\frac{(e^{ix})^4 + 1}{(e^{ix})^2}$$

6.25 Rewriting transcendental and trigonometric expressions

6.25.1 Expanding transcendental and trigonometric expressions: `texpand` `tExpand`

The `texpand` command expands exponential and trigonometric functions, like simultaneous calling:

`expexpand` (see Section 6.24.2 p.325), which, for example, expands `exp(nx)` as `exp(x)n`,

`lnexpand` (see Section 6.24.3 p.326), which, for example, expands `ln(xn)` as `n ln(x)`, and

`trigexpand` (see Section 6.23.1 p.310), which, for example, expands `sin(2x)` as `2 sin(x) cos(x)`.

`tExpand` is a synonym for `texpand`.

- `texpand` takes one argument:
expr, an expression containing transcendental or trigonometric functions.
- `texpand(expr)` expands these functions.

Examples.

- Expand `cos(x + y)`.

Input:

```
texpand(cos(x+y))
```

Output:

$$\cos x \cdot \cos y - \sin x \cdot \sin y$$

- Expand $\cos(3x)$.

Input:

`texpand(cos(3*x))`

Output:

$$4 \cos^3 x - 3 \cos x$$

- Expand $\frac{\sin(3 * x) + \sin(7 * x)}{\sin(5 * x)}$.

Input:

`texpand((sin(3*x)+sin(7*x))/sin(5*x))`

Output:

$$-\frac{2 \sin x}{(16 \cos^4 x - 12 \cos^2 x + 1) \sin x} + \frac{28 \sin x \cdot \cos^2 x}{(16 \cos^4 x - 12 \cos^2 x + 1) \sin x} -$$

$$\frac{80 \sin x \cdot \cos^4 x}{(16 \cos^4 x - 12 \cos^2 x + 1) \sin x} + \frac{64 \sin x \cdot \cos^6 x}{(16 \cos^4 x - 12 \cos^2 x + 1) \sin x}$$

Output, after a simplification with `normal(ans())`:

$$4 \cos^2 x - 2$$

- Expand $\exp(x + y)$.

Input:

`texpand(exp(x+y))`

Output:

$$e^x e^y$$

- Expand $\ln(x \times y)$.

Input:

`texpand(log(x*y))`

Output:

$$\ln y + \ln x$$

- Expand $\ln(x^n)$.

Input:

`texpand(ln(x^n))`

Output:

$$n \ln x$$

- Expand $\ln((e^2) + \exp(2 * \ln(2)) + \exp(\ln(3) + \ln(2)))$.

Input:

```
texpand(log(e^2)+exp(2*log(2))+exp(log(3)+log(2)))
```

Output:

$$6 + 2 \cdot 3$$

or input:

```
texpand(log(e^2)+exp(2*log(2)))+
lncollect(exp(log(3)+log(2)))
```

Output:

$$12$$

- Expand $\exp(x + y) + \cos(x + y) + \ln(3x^2)$.

Input:

```
texpand(exp(x+y)+cos(x+y)+ln(3*x^2))
```

Output:

$$\cos x \cdot \cos y - \sin x \cdot \sin y + e^x e^y + \ln(3) + 2 \ln|x|$$

6.25.2 Combining terms of the same type: combine

The `combine` command joins subexpressions of various types.

- `combine` takes two arguments:
 - *expr*, an expression.
 - *function*, the name of a function or class of functions. *function* can be one of `exp`, `log`, `ln`, `sin`, `cos`, or `trig`.
- `combine(expr, function)` returns the expression with subexpressions corresponding to the second argument combined.

The `combine` command can duplicate the effect of other commands.

- `combine(expr, ln)` or `combine(expr, log)` gives the same result as `lncollect(expr)` (see Section 6.24.5 p.327).
- `combine(expr, trig)` or `combine(expr, sin)` or `combine(expr, cos)` gives the same result as `tcollect(expr)` (see Section 6.23.4 p.312).

Examples.

- *Input:*

```
combine(exp(x)*exp(y)+sin(x)*cos(x)+ln(x)+ln(y), exp)
```

Output:

$$\cos x \cdot \sin x + \ln x + \ln y + e^{x+y}$$

- *Input:*

```
combine(exp(x)*exp(y)+sin(x)*cos(x)+ln(x)+ln(y),trig)
```

or:

```
combine(exp(x)*exp(y)+sin(x)*cos(x)+ln(x)+ln(y),sin)
```

or:

```
combine(exp(x)*exp(y)+sin(x)*cos(x)+ln(x)+ln(y),cos)
```

Output:

$$e^x e^y + \ln x + \ln y + \frac{\sin(2x)}{2}$$

• *Input:*

```
combine(exp(x)*exp(y)+sin(x)*cos(x)+ln(x)+ln(y),ln)
```

or:

```
combine(exp(x)*exp(y)+sin(x)*cos(x)+ln(x)+ln(y),log)
```

Output:

$$\cos x \cdot \sin x + e^x e^y + \ln(xy)$$

6.26 Fourier transformation

6.26.1 Fourier coefficients: `fourier_an` and `fourier_bn` or `fourier_cn`

Let f be a T -periodic continuous function on \mathbb{R} except perhaps at a finite number of points. One can prove that if f is continuous at x , then;

$$\begin{aligned} f(x) &= \frac{a_0}{2} + \sum_{n=1}^{+\infty} a_n \cos\left(\frac{2\pi nx}{T}\right) + b_n \sin\left(\frac{2\pi nx}{T}\right) \\ &= \sum_{n=-\infty}^{+\infty} c_n e^{\frac{2i\pi nx}{T}} \end{aligned}$$

where the coefficients a_n , b_n , $n \in N$, (or c_n , $n \in Z$) are the Fourier coefficients of f . The `fourier_an` and `fourier_bn` or `fourier_cn` commands compute these coefficients.

`fourier_an`

- `fourier_an` takes four mandatory and one optional argument:
 - *expr*, an expression depending on a variable.
 - *x*, the name of this variable.
 - *T*, the period.

- n , a non-negative integer.
- Optionally, a a real number (by default $a = 0$).
- `fourier_an(expr, x, T, n, a)` returns the Fourier coefficient a_n of a function f of variable x defined on $[a, a + T)$ by $f(x) = \text{expr}$ and such that f is periodic with period T :

$$a_n = \frac{2}{T} \int_a^{a+T} f(x) \cos\left(\frac{2\pi nx}{T}\right) dx$$

To simplify the computations, you should input `assume(n, integer)` (see Section 5.4.8 p.104) before calling `fourier_an` with an unspecified n to specify that it is an integer.

Example.

Let the function f , with period $T = 2$, be defined on $[-1, 1)$ by $f(x) = x^2$.

Input (to have the coefficient a_0):

```
fourier_an(x^2, x, 2, 0, -1)
```

Output:

$$\frac{1}{3}$$

Input (to have the coefficient a_n ($n \neq 0$)):

```
assume(n, integer)
fourier_an(x^2, x, 2, n, -1)
```

Output:

$$\frac{4(-1)^n}{n^2\pi^2}$$

fourier_bn

- `fourier_bn` takes four mandatory and one optional argument:
 - expr , and expression depending on a variable.
 - x , the name of this variable.
 - T , the period.
 - n , an integer.
 - Optionally, a a real number (by default $a = 0$).
- `fourier_bn(expr, x, T, n, a)` returns the Fourier coefficient b_n of a function f of variable x defined on $[a, a + T)$ by $f(x) = \text{expr}$ and such that f is periodic with period T :

$$b_n = \frac{2}{T} \int_a^{a+T} f(x) \sin\left(\frac{2\pi nx}{T}\right) dx$$

To simplify the computations, you should input `assume(n, integer)` (see Section 5.4.8 p.104) before calling `fourier_bn` to specify that n is an integer.

Examples.

- Let the function f , with period $T = 2$, defined on $[-1, 1)$ by $f(x) = x^2$.
Input (to get the coefficient b_n ($n \neq 0$)):

```
assume(n,integer)
fourier_bn(x^2,x,2,n,-1)
```

Output:

0

- Let the function f , with period $T = 2$, defined on $[-1, 1)$ by $f(x) = x^3$.
Input (to get the coefficient b_1):

```
fourier_bn(x^3,x,2,1,-1)
```

Output:

$$\frac{2\pi^2 - 12}{\pi^3}$$

fourier_cn

- **fourier_cn** takes four mandatory and one optional argument:
 - *expr*, and expression depending on a variable.
 - *x*, the name of this variable.
 - *T*, the period.
 - *n*, an integer.
 - Optionally, *a* a real number (by default $a = 0$).
- **fourier_cn**(*expr*, *x*, *T*, *n* [, *a*]) returns the Fourier coefficient c_n of a function f of variable x defined on $[a, a + T)$ by $f(x) = \text{expr}$ and such that f is periodic with period T :

$$c_n = \frac{1}{T} \int_a^{a+T} f(x) e^{\frac{-2i\pi nx}{T}} dx$$

To simplify the computations, you should input **assume(n,integer)** (see Section 5.4.8 p.104) before calling **fourier_cn** to specify that n is an integer.

Examples.

- Find the Fourier coefficients c_n of the periodic function f of period 2 and defined on $[-1, 1)$ by $f(x) = x^2$.
Input (to get c_0):

```
fourier_cn(x^2,x,2,0,-1)
```

Output:

$$\frac{1}{3}$$

Input (to get c_n):

```
assume(n, integer)
fourier_cn(x^2, x, 2, n, -1)
```

Output:

$$\frac{2(-1)^n}{n^2\pi^2}$$

- Find the Fourier coefficients c_n of the periodic function f , of period 2, and defined on $[0, 2)$ by $f(x) = x^2$.

Input (to have c_0):

```
fourier_cn(x^2, x, 2, 0)
```

Output:

$$\frac{4}{3}$$

Input (to get c_n):

```
assume(n, integer)
fourier_cn(x^2, x, 2, n)
```

Output:

$$\frac{\pi \cdot 2in + 2}{n^2\pi^2}$$

- Find the Fourier coefficients c_n of the periodic function f of period 2π and defined on $[0, 2\pi)$ by $f(x) = x^2$.

Input:

```
assume(n, integer)
fourier_cn(x^2, x, 2*pi, n)
```

Output:

$$\frac{\pi \cdot 2in + 2}{n^2}$$

You must also compute c_n for $n = 0$: *Input:*

```
fourier_cn(x^2, x, 2*pi, 0)
```

Output:

$$\frac{4}{3}\pi^2$$

Hence for $n = 0$, $c_0 = \frac{4\pi^2}{3}$.

Remarks.

- Input `purge(n)` (see Section 5.4.9 p.107) to remove the hypothesis done on n .
- Input `about(n)` or `assume(n)`, to know the hypothesis done on the variable n .

6.26.2 Continuous Fourier Transform: `fourier` `ifourier` `addtable`

The Fourier transform of a function f is defined by

$$F(s) = \int_{-\infty}^{+\infty} e^{-isx} f(x) dx, \quad s \in \mathbb{R}. \quad (6.5)$$

The `fourier` command computes the Fourier transform.

- `fourier` takes one mandatory argument and two optional arguments:
 - `expr`, an expression which defines a function $f(x) = \text{expr}$.
 - Optionally, x , the variable for f (by default `x`).
 - Optionally, s , the variable for the Fourier transform (by default x).
- `fourier(expr < , x, s >)` returns the Fourier transform $F(s)$. If s is not given, then x will be used.

The inverse Fourier transform, as its name implies, takes a Fourier transform $F(x)$ and returns the original function $f(x)$. It is given by:

$$f(x) = \frac{1}{2\pi} \int_{-\infty}^{+\infty} e^{isx} F(s) ds. \quad (6.6)$$

The `ifourier` command computes the inverse Fourier transform.

- `ifourier` takes one mandatory argument and two optional arguments:
 - `expr`, an expression which defines a function $F(x) = \text{expr}$.
 - Optionally, x , the variable for F (by default `x`).
 - Optionally, X , the variable for the original function f (by default x).
- `ifourier(expr < , x, X >)` returns the inverse Fourier transform $f(X)$. If X is not given, then x will be used.

Note the similarity between the definitions of the Fourier transform (equation (6.5)) and its inverse (equation (6.6)). To compute the inverse transformation of $F(s)$, it is enough to compute the Fourier transform with function $\frac{F(s)}{2\pi}$ and using the variables s and x instead of x and s , and replacing x with $-x$ in the result.

Arbitrary rational functions can be transformed.

Examples.

- Find the Fourier transform of $f(x) = \frac{x}{x^3 - 19x + 30}$.
Input:

```
F:=fourier(x/(x^3-19x+30),x,s)
```


Output:

$$\frac{1}{56}\pi\text{sign}(s)(16ie^{-2is} - 21ie^{-3is} + 5ie^{5is})$$

Input:

`ifourier(F,s,x)`

Output:

$$\frac{x}{x^3 - 19x + 30}$$

- Find the transform of $f(x) = \frac{x^2+1}{x^2-1}$:

Input:

`F:=fourier((x^2+1)/(x^2-1),x,s)`

Output:

$$2\pi(\delta(s) - \text{sign}(s)\sin s)$$

Input:

`ifourier(F,s,x)`

Output:

$$\frac{x^2 + 1}{x^2 - 1}$$

A range of other (generalized) functions and distributions can be transformed, as demonstrated in the following examples. If `fourier` does not know how to transform a function, it returns the unevaluated integral (6.5). In these cases you may try to evaluate the result using `eval`.

Examples.

- *Input:*

`fourier(3x^2+2x+1,x,s)`

Output:

$$2\pi(\delta(s) + 2i\delta(s,1) - 3\delta(s,2))$$

- *Input:*

`fourier(Dirac(x-1)+Dirac(x+1),x,s)`

Output:

$$2\cos s$$

- *Input:*

`fourier(exp(-2*abs(x-1)),x,s)`

Output:

$$\frac{4e^{-is}}{s^2 + 4}$$

• *Input:*

`fourier(atan(1/(2x^2)),x,s)`

Output:

$$\frac{2\pi e^{-\frac{|s|}{2}} \sin\left(\frac{s}{2}\right)}{s}$$

$$2\pi \sin(s/2) \exp(-\text{abs}(s)/2) / s$$

• *Input:*

`fourier(BesselJ(3,x),x,s)`

Output:

$$-\frac{s(4s^2 - 3)(-\text{isign}(s+1) + \text{isign}(s-1))}{\sqrt{-s^2 + 1}}$$

• *Input:*

`F:=fourier(sin(x)*sign(x),x,s)`

Output:

$$-\frac{2}{s^2 - 1}$$

Input:

`ifourier(F,s,x)`

Output:

$$\text{sign}(x) \sin x$$

• *Input:*

`fourier(log(abs(x)),x,s)`

Output:

$$-\frac{\pi(2\gamma\delta(s)|s| + 1)}{|s|}$$

• *Input:*

`fourier(rect(x),x,s)`

Output:

$$\frac{2 \sin\left(\frac{s}{2}\right)}{s}$$

- *Input:*

```
fourier(exp(-abs(x))*sinc(x),x,s)
```

Output:

$$\arctan(s+1) - \arctan(s-1)$$

- *Input:*

```
fourier(1/sqrt(abs(x)),x,s)
```

Output:

$$\frac{\sqrt{2}\sqrt{\pi}}{\sqrt{|s|}}$$

- *Input:*

```
F:=fourier(1/cosh(2x),x,s)
```

Output:

$$\frac{\pi}{e^{-\frac{1}{4}\pi s} + e^{\frac{1}{4}\pi s}}$$

Input:

```
ifourier(F,s,x)
```

Output:

$$\frac{2}{e^{-2x} + e^{2x}}$$

- *Input:*

```
fourier(Airy_Ai(x/2),x,s)
```

Output:

$$2e^{\frac{8}{3}is^3}$$

- *Input:*

```
F:=fourier(Gamma(1+i*x/3),x,s)
```

Output:

$$6\pi e^{-(3s+e^{-3s})}$$

Input:

```
ifourier(F,s,x)
```

Output:

$$\Gamma\left(\frac{1}{3}ix + 1\right)$$

• *Input:*

`F:=fourier(atan(x/4)/x,x,s)`

Output:

$\pi \operatorname{ugamma}(0, 4 |s|)$

Input:

`ifourier(F,s,x)`

Output:

$$\frac{\arctan\left(\frac{x}{4}\right)}{x}$$

• *Input:*

`assume(a>0)`
`fourier(exp(-a*x^2+b),x,s)`

Output:

$$\frac{\sqrt{a}\sqrt{\pi}e^{-\frac{s^2}{4a}+b}}{a}$$

The Fourier transform behaves nicely when combined with convolutions. Recall the *convolution* (see Section 15.2.8 p.1083) of two functions f and g is

$$(f * g)(x) = \int_{-\infty}^{+\infty} f(t) g(x - t) dt$$

If $\mathcal{F}(f)$ represents the Fourier transform of a function f , then the convolution theorem states

$$\mathcal{F}(f * g) = \mathcal{F}(f) \cdot \mathcal{F}(g).$$

Example.

In this example, the convolution theorem will be used to compute the convolution of $f(x) = e^{-|x|}$ with itself.

Input:

`F:=fourier(exp(-abs(x)),x,s)`

Output:

$$\frac{2}{s^2 + 1}$$

Input:

`ifourier(F^2,s,x)`

Output:

$$-x \theta(-x) e^x + x \theta(x) e^{-x} + e^{-|x|}$$

The above result is the desired convolution $(f * f)(x) = \int_{-\infty}^{+\infty} f(t) f(x - t) dt$.

Piecewise functions can be transformed if defined as

`piecewise(x < a1, f1, x < a2, f2, ..., x < an, fn, f0)`

for appropriate functions f_0, \dots, f_n and a_1, a_2, \dots, a_n are real numbers such that $a_1 < a_2 < \dots < a_n$. Inequalities may be strict or non-strict.

Example.

Input:

```
f:=piecewise(x<=-1,exp(x+1),x<=1,1,exp(2-2x))
F:=fourier(f,x,s)
```

Output:

$$\frac{3s \cos s - i s \sin s + 4 \sin s}{s(s - 2i)(s + i)}$$

You can obtain the original function f from the above result by applying `ifourier`.

Input:

```
ifourier(F,s,x)
```

Output:

$$\theta(-x-1)e^{x+1} + \theta(x+1) + \theta(x-1)e^{-2x+2} - \theta(x-1)$$

You can verify that the above expression is equal to $f(x)$ by plotting them.

Some algebraic transformations of a function behave predictably under the Fourier transform. For example, if $g(x) = f(x - a)$, then $\mathcal{F}(g)(s) = e^{-2\pi i a s} \mathcal{F}(f)(s)$. The `addtable` command lets you assign a function name to the Fourier (or Laplace, see Section 6.57.2 p.641) transform of another function name, without specifying the either function. This allows you to alter the original function and see the effect on the Fourier (or Laplace) transform.

- `addtable` takes five arguments:
 - *transform*, which can be `fourier` or `laplace` and indicates the type of transform.
 - $f(x)$, where f is a symbol representing an unspecified function of the variable x .
 - $F(s)$, where F is a symbol representing the transform of f and s is the new variable.
 - x , the variable used by f .
 - s , the variable used by F .
- `addtable(transform, f(x), F(s), x, s)` returns 1 if F is assigned as the transform of f , and 0 otherwise. In the case that F is assigned as the transform of f , then the transform (`fourier` or `laplace`) of manipulations of f will be returned in terms of F and conversely.

Examples.

- *Input:*

```
addtable(fourier,y(x),Y(s),x,s)
```

Output:

1

Input:

```
fourier(y(a*x+b),x,s)
```

Output:

$$\frac{e^{\frac{ibs}{a}} Y\left(\frac{s}{a}\right)}{|a|}$$

Input:

```
fourier(Y(x),x,s)
```

Output:

$$2\pi y(-s)$$

• *Input:*

```
addtable(fourier,g(x,t),G(s,t),x,s)
```

Output:

1

Input:

```
fourier(g(x/2,3*t),x,s)
```

Output:

$$2G(2s,3t)$$

Fourier transforms can be used for solving linear differential equations with constant coefficients. For example, to obtain a particular solution to the equation

$$y(x) + 4y^{(4)}(x) = \delta(x),$$

where δ is the Dirac delta function, you can first transform both sides of the above equation.

Input:

```
L:=fourier(y(x)+4*diff(y(x),x,4),x,s); R:=fourier(Dirac(x),x,s)
```

Output:

$$4s^4Y(s) + Y(s), 1$$

Then you can solve the equation $L = R$ for $Y(s)$. Generally, you should apply `csolve` instead of `solve`.

Input:

```
sol:=csolve(L=R,Y(s))[0]
```

Output:

$$\frac{1}{4s^4 + 1}$$

Finally, you can apply `ifourier` to obtain $y(x)$.

Input:

```
ifourier(sol,s,x)
```

Output:

$$\frac{1}{4}e^{-\frac{|x|}{2}} \left(\cos\left(\frac{|x|}{2}\right) + \sin\left(\frac{|x|}{2}\right) \right)$$

The above solution can be combined with solutions of the corresponding homogeneous equation to obtain the general solution.

6.26.3 Discrete Fourier Transform and the Fast Fourier Transform

For any integer N , the Discrete Fourier Transform (DFT) is a transformation F_N defined on the set of periodic sequences of period N ; it depends on a choice of a primitive N -th root of unity ω_N . For sequences with complex coefficients, we take:

$$\omega_N = e^{\frac{2i\pi}{N}}$$

If x is a periodic sequence of period N , defined by the vector $x = [x_0, x_1, \dots, x_{N-1}]$ then $F_N(x) = y$ is a periodic sequence of period N , defined by:

$$(F_N(x))_k = y_k = \sum_{j=0}^{N-1} x_j \omega_N^{-k \cdot j},$$

for $k = 0..N-1$.

The Discrete Fourier Transform F_N is bijective with inverse

$$F_N^{-1} = \frac{1}{N} \overline{F_{N,\omega_N}} \quad \text{on } \mathbb{C}$$

i.e.:

$$(F_{N,\omega_N}^{-1}(x))_k = \frac{1}{N} \sum_{j=0}^{N-1} x_j \omega_N^{k \cdot j}$$

The Fast Fourier Transform (FFT) is an efficient way to compute the discrete Fourier transform; faster than computing each term individually. `Xcas` implements the FFT algorithm to compute the discrete Fourier transform when the period of the sequence is a power of 2.

The `fft` command computes the discrete Fourier transform.

- `fft` takes one argument:
 x , a list or sequence regarded as one period of a periodic sequence.
- `fft(x)` returns $F_N(x)$, the discrete Fourier transform of x .
 If x has length which is a power of 2, then $F_N(x)$ is computed with the Fast Fourier Transform.

The `ifft` command computes the inverse discrete Fourier transform.

- `ifft` takes one argument:
 x , a list or sequence regarded as one period of a periodic sequence.
- `ifft(x)` returns $F_N^{-1}(x)$, the inverse discrete Fourier transform of x .
 If x has length which is a power of 2, then $F_N^{-1}(x)$ is computed with the Fast Fourier Transform.

Examples.

- *Input:*

`fft(0,1,1,0)`

Output:

`[2.0, -1.0 - i, 0.0, -1.0 + i]`

- *Input:*

`ifft([2, -1-i, 0, -1+i])`

Output:

`[0.0, 1.0, 1.0, 0.0]`

The properties of the Discrete Fourier Transform

Definitions. Let x and y be two periodic sequences of period N .

- The Hadamard product (notation \cdot) is defined by:

$$(x \cdot y)_k = x_k y_k$$

- the convolution product (notation $*$) is defined by:

$$(x * y)_k = \sum_{j=0}^{N-1} x_j y_{k-j}$$

Properties.

$$\begin{aligned} F_N(x \cdot y) &= \left(\frac{1}{N}\right) F_N(x) * F_N(y) \\ F_N(x * y) &= F_N(x) \cdot F_N(y) \end{aligned}$$

Applications

1. Value of a polynomial

Define a polynomial $P(x) = \sum_{j=0}^{N-1} c_j x^j$ by the vector of its coefficients $c := [c_0, c_1, \dots, c_{N-1}]$, where zeroes may be added so that N is a power of 2 (so the Fast Fourier Transform can be used).

- Compute the values of $P(x)$ at

$$x = a_k = \omega_N^{-k} = \exp\left(\frac{-2ik\pi}{N}\right), \quad k = 0..N-1$$

This is just the discrete Fourier transform of c since

$$P(a_k) = \sum_{j=0}^{N-1} c_j (\omega_N^{-k})^j = F_N(c)_k$$

Example.

Find the values of $P(x+x^2)$ at $x = 1, i, -1, -i$.

Input:

$$P(x) := x + x^2$$

Here the coefficients of P are $[0, 1, 1, 0]$, $N = 4$ and $\omega = \exp(2i\pi/4) = i$.

Input:

$$\text{fft}([0, 1, 1, 0])$$

Output:

$$[2.0, -1.0 - i, 0.0, -1.0 + i]$$

Hence:

- $P(1) = 2$,
- $P(-i) = P(\omega^{-1}) = -1 - i$,
- $P(-1) = P(\omega^{-2}) = 0$,
- $P(i) = P(\omega^{-3}) = -1 + i$.

- Compute the values of $P(x)$ at

$$x = b_k = \omega_N^k = \exp\left(\frac{2ik\pi}{N}\right), \quad k = 0..N-1$$

This is N times the inverse fourier transform of c since

$$P(a_k) = \sum_{j=0}^{N-1} c_j (\omega_N^k)^j = N F_N^{-1}(c)_k$$

Example.

Use this method to find the values of $P(x+x^2)$ at $x = 1, i, -1, -i$.

Input:

$$P(x) := x + x^2$$

Again, the coefficients of P are $[0, 1, 1, 0]$, $N = 4$ and $\omega = \exp(2i\pi/4) = i$.

Input:

$$4 * \text{ifft}([0, 1, 1, 0])$$

Output:

$$[2.0, -1.0 + i, 0.0, -1.0 - i]$$

Hence:

- $P(1) = 2$,
- $P(i) = P(\omega^1) = -1 + i$,
- $P(-1) = P(\omega^2) = 0$,
- $P(-i) = P(\omega^3) = -1 - i$.

You find of course the same values as above.

2. Trigonometric interpolation

Let f be periodic function of period 2π and let $f_k = f(2k\pi/N)$ for $k = 0..(N-1)$. Find a trigonometric polynomial p that interpolates f at $x_k = 2k\pi/N$, that is find $p_j, j = 0..N-1$ such that

$$p(x) = \sum_{j=0}^{N-1} p_j x^j, \quad p(x_k) = f_k$$

Replacing x_k by its value in $p(x)$ we get:

$$\sum_{j=0}^{N-1} p_j \exp(i \frac{j2k\pi}{N}) = f_k$$

In other words, (f_k) is the inverse DFT of (p_k) , hence

$$(p_k) = \frac{1}{N} F_N((f_k))$$

If the function f is real, $p_{-k} = \bar{p}_k$, hence depending whether N is even or odd:

$$p(x) = p_0 + 2\Re\left(\sum_{k=0}^{\frac{N}{2}-1} p_k \exp(ikx)\right) + \Re\left(p_{\frac{N}{2}} \exp(i \frac{Nx}{2})\right)$$

if N is even and

$$p(x) = p_0 + 2\Re\left(\sum_{k=0}^{\frac{N-1}{2}} p_k \exp(ikx)\right)$$

if N is odd.

3. Fourier series

Let f be a periodic function of period 2π and let $y_k = f(x_k)$ where $x_k = \frac{2k\pi}{N}$ for $k = 0..N-1$. Suppose that the Fourier series of f converges to f (this will be the case if for example f is continuous). If N is large, a good approximation of f will be given by:

$$\sum_{-\frac{N}{2} \leq n < \frac{N}{2}} c_n \exp(inx)$$

Hence we want a numeric approximation of

$$c_n = \frac{1}{2\pi} \int_0^{2\pi} f(t) \exp(-int) dt$$

The numeric value of the integral $\int_0^{2\pi} f(t) \exp(-int) dt$ can be computed by the trapezoidal rule (note that the Romberg algorithm would not work here because the Euler Mac Laurin development has its coefficients equal to zero, since the integrated function is periodic, hence all its derivatives have the same value at 0 and at 2π). If \tilde{c}_n is the numeric value of c_n obtained by the trapezoidal rule, then

$$\tilde{c}_n = \frac{1}{2\pi} \frac{2\pi}{N} \sum_{k=0}^{N-1} y_k \exp(-2i \frac{nk\pi}{N}), \quad -\frac{N}{2} \leq n < \frac{N}{2}$$

Indeed, since $x_k = 2k\pi/N$ and $f(x_k) = y_k$:

$$\begin{aligned} f(x_k) \exp(-inx_k) &= y_k \exp(-2i \frac{nk\pi}{N}), \\ f(0) \exp(0) = f(2\pi) \exp(-2i \frac{nN\pi}{N}) &= y_0 = y_N \end{aligned}$$

Hence:

$$[\tilde{c}_0, \dots, \tilde{c}_{\frac{N}{2}-1}, \tilde{c}_{\frac{N}{2}+1}, \dots, \tilde{c}_{N-1}] = \frac{1}{N} F_N([y_0, y_1, \dots, y_{(N-1)}])$$

since

- if $n \geq 0$, $\tilde{c}_n = y_n$
- if $n < 0$ $\tilde{c}_n = y_{n+N}$
- $\omega_N = \exp(\frac{2i\pi}{N})$, so $\omega_N^n = \omega_N^{n+N}$

Properties.

- The coefficients of the trigonometric polynomial that interpolates f at $x = 2k\pi/N$ are

$$p_n = \tilde{c}_n, \quad -\frac{N}{2} \leq n < \frac{N}{2}$$

- If f is a trigonometric polynomial of degree $m \leq \frac{N}{2}$, then

$$f(t) = \sum_{k=-m}^{m-1} c_k \exp(2ik\pi t)$$

the trigonometric polynomial that interpolates f is f itself, the numeric approximation of the coefficients are in fact exact ($\tilde{c}_n = c_n$).

- More generally, you can compute $\tilde{c}_n - c_n$.
Suppose that f is equal to its Fourier series, i.e. that:

$$f(t) = \sum_{m=-\infty}^{+\infty} c_m \exp(2i\pi mt), \quad \sum_{m=-\infty}^{+\infty} |c_m| < \infty$$

Then:

$$f(x_k) = f\left(\frac{2k\pi}{N}\right) = y_k = \sum_{m=-\infty}^{+\infty} c_m \omega_N^{km}, \quad \tilde{c}_n = \frac{1}{N} \sum_{k=0}^{N-1} y_k \omega_N^{-kn}$$

Replace y_k by its value in \tilde{c}_n :

$$\tilde{c}_n = \frac{1}{N} \sum_{k=0}^{N-1} \sum_{m=-\infty}^{+\infty} c_m \omega_N^{km} \omega_N^{-kn}$$

If $m \neq n \pmod{N}$, ω_N^{m-n} is an N -th root of unity different from 1, hence:

$$\omega_N^{(m-n)N} = 1, \quad \sum_{k=0}^{N-1} \omega_N^{(m-n)k} = 0$$

Therefore, if $m - n$ is a multiple of N ($m = n + l \cdot N$) then $\sum_{k=0}^{N-1} \omega_N^{k(m-n)} = N$, otherwise $\sum_{k=0}^{N-1} \omega_N^{k(m-n)} = 0$. By reversing the two sums, you get

$$\begin{aligned} \tilde{c}_n &= \frac{1}{N} \sum_{m=-\infty}^{+\infty} c_m \sum_{k=0}^{N-1} \omega_N^{k(m-n)} \\ &= \sum_{l=-\infty}^{+\infty} c_{(n+l \cdot N)} \\ &= \dots c_{n-2 \cdot N} + c_{n-N} + c_n + c_{n+N} + c_{n+2 \cdot N} + \dots \end{aligned}$$

Conclusion: if $|n| < N/2$, then $\tilde{c}_n - c_n$ is a sum of c_j with large indices (at least $N/2$ in absolute value), hence is small (depending on the rate of convergence of the Fourier series).

Example.

Input:

```
f(t):=cos(t)+cos(2*t)
x:=f(2*k*pi/8)$(k=0..7)
```

Output:

$$2, \frac{\sqrt{2}}{2}, -1, -\frac{\sqrt{2}}{2}, 0, -\frac{\sqrt{2}}{2}, -1, \frac{\sqrt{2}}{2}$$

Input:

$$\text{fft}(\mathbf{x})$$

Output:

$$[0.0, 4.0, 4.0, 0.0, 0.0, 0.0, 4.0, 4.0]$$

Dividing by $N = 8$, you get

$$c_0 = 0, c_1 = 0.5, c_2 = 0.5, c_3 = 0.0, \\ c_{-4} = 0.0, c_{-3} = 0.0, c_{-2} = 0.5, c_{-1} = 0.5$$

Hence $b_k = 0$ and $a_k = c_{-k} + c_k$ equals 1 for $k = 1, 2$ and 0 otherwise.

4. Convolution Product

If $P(x) = \sum_{j=0}^{n-1} a_j x^j$ and $Q(x) = \sum_{j=0}^{m-1} b_j x^j$ are given by the vectors of their coefficients $a = [a_0, a_1, \dots, a_{n-1}]$ and $b = [b_0, b_1, \dots, b_{m-1}]$, you can compute the product of these two polynomials using the DFT. The product of polynomials is the convolution product of the periodic sequence of their coefficients if the period is greater or equal to $(n+m)$. Therefore we complete a (resp. b) with $m+p$ (resp. $n+p$) zeros, where p is chosen such that $N = n+m+p$ is a power of 2. If $a = [a_0, a_1, \dots, a_{n-1}, 0, \dots, 0]$ and $b = [b_0, b_1, \dots, b_{m-1}, 0, \dots, 0]$, then:

$$P(x)Q(x) = \sum_{j=0}^{n+m-1} (a * b)_j x^j$$

If you know $F_N(a)$ and $F_N(b)$, then $a * b = F_N^{-1}(F_N(a) \cdot F_N(b))$, since

$$F_N(x * y) = F_N(x) \cdot F_N(y)$$

6.26.4 An exercise with fft

Given temperatures T at time t , in degrees Celcius:

t	0	3	6	9	12	15	19	21
T	11	10	17	24	32	26	23	19

What was the temperature at 13h45 ?

Here $N = 8 = 2 * m$. The interpolation polynomial is

$$p(t) = \frac{1}{2} p_{-m} \left(\exp(-2i \frac{\pi m t}{24}) + \exp(2i \frac{\pi m t}{24}) \right) + \sum_{k=-m+1}^{m-1} p_k \exp(2i \frac{\pi k t}{24})$$

and

$$p_k = \frac{1}{N} \sum_{j=0}^{N-1} T_j \exp(2i \frac{\pi k j}{N})$$

Input:

```
q:=1/8*fft([11,10,17,24,32,26,23,19])
```

Output:

```
[20.25,-4.48115530061+1.72227182413i,0.375+0.875i,
 -0.768844699385+0.222271824132i,0.5,
 -0.768844699385-0.222271824132i,
 0.375-0.875i,-4.48115530061-1.72227182413i]
```

hence:

- $p_0 = 20.25$
- $p_1 = -4.48115530061 + 1.72227182413i = \overline{p_{-1}},$
- $p_2 = 0.375 + 0.875i = \overline{p_{-2}},$
- $p_3 = -0.768844699385 + 0.222271824132i = \overline{p_{-3}},$
- $p_4 = 0.5$

Indeed

$$q = [q_0, \dots, q_{N-1}] = [p_0, \dots, p_{\frac{N}{2}-1}, p_{-\frac{N}{2}}, \dots, p_{-1}] = \frac{1}{N} F_N([y_0, \dots, y_{N-1}]) = \frac{1}{N} \text{fft}(y)$$

Input:

```
pp:=[q[4],q[5],q[6],q[7],q[0],q[1],q[2],q[3]]
```

Here, $p_k = pp[k+4]$ for $k = -4 \dots 3$. It remains to compute the value of the interpolation polynomial at point $t_0 = 13.75 = 55/4$.

Input:

```
t0(j):=exp(2*i*pi*(13+3/4)/24*j)
T0:=1/2*pp[0]*(t0(4)+t0(-4))+sum(pp[j+4]*t0(j),j,-3,3)
evalf(re(T0))
```

Output:

```
29.4863181684
```

The temperature is predicted to be equal to 29.49 degrees Celsius.

Remark.

Using the Lagrange interpolation polynomial (the polynomial is not periodic):

Input:

```
l1:=[0,3,6,9,12,15,18,21]
l2:=[11,10,17,24,32,26,23,19]
subst(lagrange(l1,l2,13+3/4),x=13+3/4)
evalf(ans())
```

Output:

```
30.1144061688
```

6.27 Polynomials

6.27.1 Polynomials of a single variable: `poly1`

A polynomial of one variable is represented either by a symbolic expression or by the list of its coefficients in decreasing powers order (dense representation). In the latter case, to avoid confusion with other kinds of lists:

- `poly1[...]` is used as delimiters for inputs and for text form output.
- `||...||` is used for `Xcas` output.

Note that polynomials represented as lists of coefficients are always written in decreasing powers order even if `increasing power` is checked in `cas` configuration (see Section 3.5.7 p.73).

6.27.2 Polynomials of several variables: `%%{ %%}`

A polynomial of several variables can be represented in different ways:

- by a symbolic expression.
- by a dense recursive 1-d representation like above.
- by a sum of monomials with non-zero coefficients (distributed sparse representation).

A monomial with several variables is represented by a coefficient and a list of integers (interpreted as powers of a variable list). The delimiters for monomials are `%%{` and `%%}`.

For example $3x^2y$ is represented by `%%{3,[2,1]%%}` with respect to the variable list `[x,y]`, and $2x^3y^2z - 5xz$ is represented by `%%{2,[3,2,1]%%} - %%{5,[1,0,1]%%}` with respect to the variable list `[x,y,z]`.

For a sparse representation, a single variable polynomial can be regarded as a multivariate polynomial with one variable.

6.27.3 Apply a function to the internal sparse format of a polynomial: `map`

The `map` command can apply a function to the coefficients of a polynomial written in internal sparse format. (See Section 6.40.28 p.484 for other uses of `map`.)

- `map` takes two arguments:
 - P , a polynomial of k variables in internal sparse format.
 - f , a function of $k + 1$ variables.
- `map(P, f)` applies f to the coefficients of P ; namely, it returns a polynomial which replaces each term `%%{ $a, [n_1, \dots, n_k]$ %%}` in P by `%%{ $f(a, n_1, \dots, n_k), [n_1, \dots, n_k]$ %%}`

Example.

Input:

```
map(%{{2,[2,1]}} + %{{3,[1,4]}},(a,b,c)->a*b*c)
```

Output:

```
%{{4,[2,1]}} + %{{12,[1,4]}}
```

6.27.4 Converting to a symbolic polynomial: `r2e` `poly2symb`

The `r2e` command converts lists into symbolic polynomials.

`poly2symb` is a synonym for `r2e` here.

For one-variable polynomials:

- `r2e` takes one mandatory argument and one optional argument:
 - L , a list of coefficients of a polynomial (in decreasing order),
 - x , a symbolic variable name (by default x).
- `r2e(L , x)` returns the corresponding polynomial with the given variable.

Example:

Input:

```
r2e([1,0,-1],x)
```

or:

```
r2e([1,0,-1])
```

or:

```
poly2symb([1,0,-1],x)
```

Output:

$$xx - 1$$

For sparse multivariate polynomials:

- `r2e` takes two arguments:
 - S , a sum of monomials of the form `%{{coeff,[n1,...nk]}}`
 - $vars$, a vector of symbolic variables.
- `r2e(S , $vars$)` returns the corresponding polynomial as an expression with the given variables

Examples:

- *Input:*

```
poly2symb(%{{1,[2]}}+%{{-1,[0]}},[x])
```

or:


```
r2e({1, [2]}+{-1, [0]}, [x])
```

Output:

$$x^2 - 1$$

- *Input:*

```
r2e({1, [2, 0]}+{-1, [1, 1]}+{2, [0, 1]}, [x, y])
```

or:

```
poly2symb({1, [2, 0]}+{-1, [1, 1]}+{2, [0, 1]}, [x, y])
```

Output:

$$x^2 - xy + 2y$$

6.27.5 Converting from a symbolic polynomial: `e2r` `symb2poly`

The `e2r` command converts a symbolic polynomial into a list (for single variable polynomials) or a sum of monomials.

`symb2poly` is a synonym for `e2r`.

- `e2r` takes two arguments:
 - P , a symbolic polynomial.
 - $vars$, the variable name (for one variable polynomials) or a list of variable names (for multivariable polynomials).
For one variable polynomials, this is optional and defaults to `x`.
- `e2r(P, vars)` returns:
 - the representation of the polynomial as a list of coefficients written in decreasing order, if $vars$ is a variable name.
 - a sum of monomials (sparse representation of multivariate polynomials) if $vars$ is a list.

Examples:

- *Input:*

```
e2r(x^2-1)
```

or:

```
symb2poly(x^2-1)
```

or:

```
symb2poly(y^2-1, y)
```

or:

```
e2r(y^2-1,y)
```

Output:

```
[[1,0,-1]]
```

• *Input:*

```
e2r(x^2-x*y+y, [x,y])
```

or:

```
symb2poly(x^2-x*y+2*y, [x,y])
```

Output:

```
%%%{1,[2,0]%%}%+%%%{-1,[1,1]%%}%+%%%{2,[0,1]%%}%
```

6.27.6 Transforming a polynomial in internal format into a list, and conversely: `convert`

The `convert` command does many conversions (see Section 6.23.26 p.323). Among other things, it can convert between a polynomial in internal sparse multivariate format and a list representing the polynomial.

To convert from a polynomial in internal sparse multivariate format to a list:

- `convert` takes one mandatory argument and one optional argument:
 - P , a polynomial written in internal sparse multivariate format (see Section 6.27.2 p.351).
 - Optionally, `list`.
- `convert(P , list)` returns a list representing the polynomial.

Example.

Input:

```
p:= symb2poly(x^2 - x*y + 2y, [x,y])
```

Output:

```
%%%{1,[2,0]%%}%+%%%{-1,[1,1]%%}%+%%%{2,[0,1]%%}%
```

Input:

```
l:= convert(p,list)
```

or:

```
l:= convert(p)
```

Output:

$$\begin{bmatrix} 1 & [2, 0] \\ -1 & [1, 1] \\ 2 & [0, 1] \end{bmatrix}$$

which is a list of the coefficients followed by a list of the variable powers.

To convert from a list representing a polynomial to the polynomial in internal sparse multivariate format:

- `convert` takes two arguments:
 - L , a list representing a polynomial.
 - `polynom`.
- `convert(L , polynom)` returns the polynomial in internal sparse multivariate format (see Section 6.27.2 p.351).

Example.

Input (1 from above):

```
1:=[[1,[2,0]],[-1,[1,1]],[2,[0,1]]]
```

Output:

$$\begin{bmatrix} 1 & [2, 0] \\ -1 & [1, 1] \\ 2 & [0, 1] \end{bmatrix}$$

Input:

```
convert(1,polynom)
```

Output:

```
%%{1,[2,0]}+%%{-1,[1,1]}+%%{2,[0,1]}
```

6.27.7 Coefficients of a polynomial: `coeff` `coeffs`

The `coeff` command finds the coefficients of a specific degree of a polynomial. `coeffs` is a synonym for `coeff`.

- `coeff` takes two mandatory and one optional argument:
 - P , the polynomial.
 - $vars$, the name of the variable (or the list of the names of variables).
 - Optionally, n , the degree (or the list of the degrees of the variables).
- `coeff($Pvars$ $\langle, n \rangle$)` returns the n th degree coefficient of P , or if n is not specified, the list of the coefficients of P , including 0 in the univariate dense case and excluding 0 in the multivariate sparse case.

Examples.

- *Input:*

```
coeff(-x^4+3*x*y^2+x,x,1)
```

Output:

$$3y^2 + 1$$

- *Input:*

```
coeff(-x^4+3x*y^2+x,y,2)
```

Output:

$$3x$$

- *Input:*

```
coeff(-x^4+3x*y^2+x,[x,y],[1,2])
```

Output:

$$3$$

6.27.8 Polynomial degree: degree

The **degree** command finds the degree of a polynomial.

- **degree** takes one argument:
 P , a polynomial given by its symbolic representation or by the list of its coefficients.
- **degree**(P) returns the degree of P (the highest degree of its non-zero monomials).

Examples.

- *Input:*

```
degree(x^3+x)
```

Output:

$$3$$

- *Input:*

```
degree([1,0,1,0])
```

Output:

$$3$$

6.27.9 Polynomial valuation: valuation ldegree

The *valuation* of a polynomial is the lowest degree of its non-zero monomials. The `valuation` command finds the valuation of a polynomial. `ldegree` is a synonym for `valuation`.

- `valuation` takes one argument:
 P , a polynomial given by a symbolic expression or by the list of its coefficients.
- `valuation(P)` returns the valuation of P .

Examples.

- *Input:*

```
valuation(x^3+x)
```

Output:

1

- *Input:*

```
valuation([1,0,1,0])
```

Output:

1

6.27.10 Leading coefficient of a polynomial: lcoeff

The `lcoeff` command finds the leading coefficient of a polynomial; that is, the coefficient of the monomial of highest degree.

- `lcoeff` takes one mandatory argument and one optional argument:
 - P , a polynomial given by a symbolic expression or by its list of coefficients.
 - Optionally, x , a variable name (by default x).
- `lcoeff($P \langle x \rangle$)` returns the leading coefficient of P .

Examples.

- *Input:*

```
lcoeff([2,1,-1,0])
```

Output:

2

- *Input:*

```
lcoeff(3*x^2+5*x,x)
```

Output:

3

- *Input:*

`lcoeff(3*x^2+5*x*y^2,y)`

Output:

$5x$

6.27.11 Trailing coefficient degree of a polynomial: `tcoeff`

The `tcoeff` command finds the trailing coefficient of a polynomial; that is, the coefficient of the monomial of lowest degree.

- `tcoeff` takes one mandatory argument and one optional argument:
 - P , a polynomial given by a symbolic expression or by its list of coefficients.
 - Optionally x , a variable name (by default x).
- `tcoeff($P\langle x \rangle$)` returns the trailing coefficient of P .

Examples.

- *Input:*

`tcoeff([2,1,-1,0])`

Output:

-1

- *Input:*

`tcoeff(3*x^2+5*x,x)`

Output:

5

- *Input:*

`tcoeff(3*x^2+5*x*y^2,y)`

Output:

$3x^2$

6.27.12 Evaluating polynomials: peval polyEval

The `peval` command evaluates polynomials.

`polyEval` is a synonym for `peval`.

- `peval` takes two arguments:
 - P , a polynomial given by the list of its coefficients.
 - a , a real number.
- `peval(P, a)` returns the exact or numeric value of $P(a)$, calculated using Horner's method.

Examples.

- *Input:*

```
peval([1,0,-1],sqrt(2))
```

Output:

$$\sqrt{2}\sqrt{2} - 1$$

then input:

```
normal(sqrt(2)*sqrt(2)-1)
```

Output:

$$1$$

- *Input:*

```
peval([1,0,-1],1.4)
```

Output:

$$0.96$$

6.27.13 Factoring x^n in a polynomial: factor_xn

The `factor_xn` command factors the largest power of the variable out of a polynomial, writing it as the product of a monomial of largest degree and a rational function having a non-zero finite limit at infinity.

- `factor_xn` takes one argument:
 - P , a polynomial.
- `factor_xn(P)` returns P written as the product of its monomial of largest degree with a rational function having a non-zero finite limit at infinity.

Example.

Input:

```
factor_xn(-x^4+3)
```

Output:

$$x^4(-1 + 3x^{-4})$$

6.27.14 GCD of the coefficients of a polynomial: content

The *content* of a polynomial is the GCD (greatest common divisor) of its coefficients. The `content` command computes the content of a polynomial.

- `content` takes one argument:
 P , a polynomial given by a symbolic expression or by the list of its coefficients.
- `content(P)` returns the content of P .

Example.

Input:

```
content(6*x^2-3*x+9)
```

or:

```
content([6,-3,9],x)
```

Output:

3

6.27.15 Primitive part of a polynomial: primpart

The primitive part of a polynomial is the polynomial divided by its content (the greatest common divisor of its coefficients). The `primpart` command computes the primitive part of a polynomial.

- `primpart` takes one argument:
 P , a polynomial given by a symbolic expression or by the list of its coefficients.
- `primpart(P)` returns the primitive part of P .

Example.

Input:

```
primpart(6x^2-3x+9)
```

or:

```
primpart([6,-3,9],x)
```

Output:

$2x^2 - x + 3$

6.27.16 Factoring: collect

The `collect` command factors polynomials over their coefficient fields or extensions of the fields.

- `collect` takes one mandatory and one optional argument:
 - P , a polynomial or a list of polynomials.
 - Optionally, α , a number, such as \sqrt{n} , determining an extension field to the field of coefficients of P .
- `collect(P, α)` returns the factored form of the polynomial (or list of polynomials), where the factorization is done over the field of coefficients (such as \mathbb{Q}) or the smallest extension field containing α (e.g. $\mathbb{Q}[\alpha]$). In complex mode (see Section 3.5.7 p.73), the field is complexified.

The `factor` command (see 6.12.10) will also factor polynomials over their coefficient fields (or extensions of it), but will further factor each factor of degree 2 if `Sqrt` is checked in the `cas` configuration.

Examples.

- Factor $x^2 - 4$ over the integers, *Input*:

```
collect(x^2-4)
```

Output (in real mode):

$$(x - 2)(x + 2)$$

- Factor $x^2 + 4$ over the integers: *Input*:

```
collect(x^2+4)
```

Output (in real mode):

$$x^2 + 4$$

Output (in complex mode):

$$(x + 2i)(x - 2i)$$

- Factor $x^2 - 2$ over the rationals: *Input*:

```
collect(x^2-2)
```

Output (in real mode):

$$x^2 - 2$$

But if you input:

```
collect(sqrt(2)*(x^2-2))
```

you get: *Output:*

$$\sqrt{2} (x - \sqrt{2}) (x + \sqrt{2})$$

- Factor $x^3 - 2x^2 + 1$ and $x^2 - x$ over the rationals. *Input:*

```
collect([x^3-2*x^2+1,x^2-x])
```

Output:

$$[(x - 1) (x^2 - x - 1), x (x - 1)]$$

but:

Input:

```
collect((x^3-2*x^2+1)*sqrt(5))
```

Output:

$$\sqrt{5} \left(x + \frac{-\sqrt{5}-1}{2} \right) (x-1) \left(x + \frac{\sqrt{5}-1}{2} \right)$$

or:

Input:

```
collect(x^3-2*x^2+1,sqrt(5))
```

Output:

$$\left(x + \frac{-\sqrt{5}-1}{2} \right) (x-1) \left(x + \frac{\sqrt{5}-1}{2} \right)$$

6.27.17 Square-free factorization: `sqrfree`

The `sqrfree` command provides squarefree factorizations of polynomials; that is, it factors a polynomial as a product of powers of coprime factors, where each factor has roots of multiplicity 1 (in other words, a factor and its derivative are coprime).

- `sqrfree` takes one argument:
 P , a polynomial.
- `sqrfree(P)` returns the squarefree factorization of P .

Examples.

- *Input:*

```
sqrfree((x^2-1)*(x-1)*(x+2))
```

Output:

$$(x^2 + 3x + 2) (x - 1)^2$$

- *Input:*

```
sqrfree((x^2-1)^2*(x-1)*(x+2)^2)
```

Output:

$$(x^2 + 3x + 2)^2 (x - 1)^3$$

6.27.18 List of factors: factors

The `factors` command provides the factors of a polynomial as a list.

- `factors` takes one argument:
 P , a polynomial or a list of polynomials.
- `factors(P)` returns a list containing the factors of P and their exponents, or a list of such lists.

Examples.

- *Input:*

```
factors(x^2+2*x+1)
```

Output:

 $[x + 1, 2]$

- *Input:*

```
factors(x^4-2*x^2+1)
```

Output:

 $[x - 1, 2, x + 1, 2]$

- *Input:*

```
factors([x^3-2*x^2+1, x^2-x])
```

Output:

$$\begin{bmatrix} x-1 & 1 & x^2-x-1 & 1 \\ x & 1 & x-1 & 1 \end{bmatrix}$$

- *Input:*

```
factors([x^2, x^2-1])
```

Output:

 $[[x, 2], [x - 1, 1, x + 1, 1]]$
6.27.19 Evaluating a polynomial: horner

The `horner` command uses Horner's method to evaluate polynomials.

- `horner` takes two arguments:
 - P , a polynomial given by its symbolic expression or by the list of its coefficients.
 - a , a number.
- `horner(P, a)` returns the value $P(a)$, computed using Horner's method.

Example.*Input:*

```
horner(x^2-2*x+1,2)
```

or:

```
horner([1,-2,1],2)
```

Output:

1

6.27.20 Rewriting in terms of the powers of (x-a): ptayl

The `ptayl` command finds the Taylor expansion for a polynomial (which will be finite).

- `ptayl` takes two arguments:
 - P , a polynomial given by a symbolic expression or by the list of its coefficients.
 - a , a number.
- `ptayl(P, a)` returns the polynomial T such that $P(x) = T(x - a)$.

Examples.

- *Input:*

```
ptayl(x^2+2*x+1,2)
```

Output, the polynomial T:

$$x^2 + 6x + 9$$

- *Input:*

```
ptayl([1,2,1],2)
```

Output:

$$[1, 6, 9]$$

i.e.; $x^2 + 2x + 1 = (x - 2)^2 + 6(x - 2) + 9$.

6.27.21 Computing with the exact root of a polynomial: rootof

The `rootof` command finds the value of one polynomial at a root of another.

- `rootof` takes two arguments:
 P and Q , two polynomials given by the lists of their coefficients.
- `rootof(P, Q)` gives the value $P(\alpha)$ where α is the root of Q with largest real part (and largest imaginary part in case of equality).

In exact computations, `Xcas` will rewrite rational evaluations of `rootof` as a unique `rootof` with $\text{degree}(P) < \text{degree}(Q)$. If the resulting `rootof` is the solution of a second degree equation, it will be simplified.

Example.

Let α be the root with largest imaginary part of $Q(x) = x^4 + 10x^2 + 1$ (all roots of Q have real part equal to 0).

- Compute $\frac{1}{\alpha}$.

Input:

```
normal(1/rootof([1,0],[1,0,10,0,1]))
```

$P(x) = x$ is represented by $[1,0]$ and α by `rootof([1,0],[1,0,10,0,1])`.

Output:

$$-i(-\sqrt{2} + \sqrt{3})$$

- Compute α^2 .

Input:

```
normal(rootof([1,0],[1,0,10,0,1])^2)
```

or (since $P(x) = x^2$ is represented by $[1,0,0]$):

Input:

```
normal(rootof([1,0,0],[1,0,10,0,1]))
```

Output:

$$-2\sqrt{6} - 5$$

6.27.22 Exact roots of a polynomial: roots

The `roots` command finds roots of polynomials with their multiplicities

- `roots` takes one mandatory and one optional argument:
 - P , a symbolic polynomial expression.
 - Optionally, x , the name of the variable (the default is \mathbf{x}).
- `roots(P, x)` returns a 2 column matrix: each row is the list consisting of a root of P and its multiplicity.

Examples.

- Find the roots of $P(x) = x^5 - 2x^4 + x^3$.

Input:

```
roots(x^5-16*x^4+x^3)
```

Output:

$$\begin{bmatrix} 3\sqrt{7}+8 & 1 \\ -3\sqrt{7}+8 & 1 \\ 0 & 3 \end{bmatrix}$$

- Find the roots of $x^{10} - 15x^8 + 90x^6 - 270x^4 + 405x^2 - 243 = (x^2 - 3)^5$.

Input:

```
roots(x^10-15*x^8+90*x^6-270*x^4+405*x^2-243)
```

Output:

$$\begin{bmatrix} \sqrt{3} & 5 \\ -\sqrt{3} & 5 \end{bmatrix}$$

- Find the roots of $t^3 - 1$.

Input:

```
roots(t^3-1,t)
```

Output:

$$\begin{bmatrix} 1 & 1 \\ \frac{i\sqrt{3}-1}{2} & 1 \\ \frac{-i\sqrt{3}-1}{2} & 1 \end{bmatrix}$$

6.27.23 Coefficients of a polynomial defined by its roots:**pcoeff pcoef**

The **pcoeff** command reconstructs a polynomial from its roots.

pcoef is a synonym for **pcoeff**.

- pcoeff** takes one argument:
roots, a list of the roots of a polynomial P .
- pcoeff(roots)** returns the monic polynomial having these roots, represented as the list of its coefficients in decreasing order.

Example.*Input:*

```
pcoef([1,2,0,0,3])
```

Output:

```
[1,-6,11,-6,0,0]
```

i.e. $(x-1)(x-2)(x^2)(x-3) = x^5 - 6x^4 + 11x^3 - 6x^2$.

6.27.24 Truncating to order n : `truncate`

The `truncate` command truncates a polynomial; i.e., it removes higher order terms.

- `truncate` takes two arguments:
 - P , a polynomial.
 - n , an integer.
- `truncate(P, n)` returns P truncated to order n ; i.e., all terms of order greater or equal to $n + 1$ are removed.

`truncate` may be used to transform a series expansion into a polynomial or to compute a series expansion step by step.

Examples.

- *Input:*

```
truncate((1+x+x^2/2)^3,4)
```

Output:

$$\frac{9x^4 + 16x^3 + 18x^2 + 12x + 4}{4}$$

- *Input:*

```
truncate(series(sin(x)),4)
```

Output:

$$\frac{-x^3 + 6x}{6}$$

Note that the returned polynomial is normalized.

6.27.25 Converting a series expansion into a polynomial: `convert` `convertir`

The `convert` command (see Section 6.23.26 p.323), with the option `polynom`, converts a series (see Section 6.36.2 p.439) into a polynomial. It should be used for operations like drawing the graph of the Taylor series of a function near a point.

For this purpose:

- `convert` takes two arguments:
 - *series*, a series.
 - `polynom`, the option.
- `convert(series, polynom)` returns *series* with the `order_size` function replaced by 0.

Examples.

- *Input:*

```
convert(taylor(sin(x)),polynom)
```

Output:

$$x - \frac{x^3}{6} + \frac{x^5}{120}$$

- *Input:*

```
convert(series(sin(x),x=0,6),polynom)
```

Output:

$$x - \frac{x^3}{6} + \frac{x^5}{120}$$

6.27.26 Random polynomial: randpoly randPoly

The **randpoly** command finds random polynomials.

randPoly is a synonym for **randpoly**.

- **randpoly** takes two optional arguments:
 - Optionally x , the name of a variable (by default **x**).
 - Optionally n , an integer (by default 10).

The order of the arguments is not important.

- **randpoly**($\langle x \rangle \langle n \rangle$) returns a monic polynomial in the variable x of degree n , having as coefficients random integers evenly distributed on $-99..+99$.

Examples.

- *Input:*

```
randpoly(t,4)
```

Output (for example):

$$t^4 + 86t^3 - 97t^2 - 82t + 7$$

- *Input:*

```
randpoly(4)
```

Output (for example):

$$x^4 - 27x^3 + 26x^2 - 89x + 63$$

- *Input:*

```
randpoly(4,u)
```

Output (for example):

$$u^4 - 49u^3 - 86u^2 - 64u - 30$$

6.27.27 Changing the order of variables: reorder

The **reorder** command rewrites an expression, based on the priority of variables.

- **reorder** takes two arguments:
 - *expr*, an expression.
 - *vars*, a vector of variable names.
- **reorder**(*expr*,*vars*) expands *expr* according to the order of variables given in *vars*.

Example.

Input:

```
reorder(x^2+2*x*a+a^2+z^2-x*z,[a,x,z])
```

Output:

$$a^2 + 2ax + x^2 - xz + z^2$$

Warning.

The variables must be symbolic (if not, purge them (see Section 5.4.8 p.104) before calling **reorder**).

6.27.28 Random lists: ranm

The **ranm** command finds lists of random integers.

- **ranm** takes one argument:
 - n*, an integer.
- **ranm**(*n*) returns a list of *n* random integers (between -99 and +99). This list can be seen as the coefficients of an univariate polynomial of degree *n* – 1.

(See also Section 9.3.16 p.757)

Example.

Input:

```
ranm(3)
```

Output (for example):

```
[70, 22, 42]
```

6.27.29 Lagrange polynomial: lagrange interp

The **lagrange** command finds the Lagrange polynomial which interpolates given data.

interp is a synonym for **lagrange**.

- **lagrange** takes two mandatory arguments and one optional argument:

- l_1 and l_2 , two lists of the same size. These can be given as a matrix with two rows.
The first list (resp. row) corresponds to the abscissa values x_k ($k = 1..n$), and the second list (resp. row) corresponds to ordinate values y_k ($k = 1..n$).
- Optionally x , the name of a variable (by default \mathbf{x}).
- **lagrange**($l_1, l_2 \langle, x \rangle$) returns a polynomial expression P with respect to x of degree $\mathbf{n}-1$, such that $P(x_i) = y_i$.

Examples.

- *Input:*

```
lagrange([[1,3],[0,1]])
```

or:

```
lagrange([1,3],[0,1])
```

Output:

$$\frac{x-1}{2}$$

since $\frac{x-1}{2} = 0$ for $x = 1$ and $\frac{x-1}{2} = 1$ for $x = 3$.

- *Input:*

```
lagrange([1,3],[0,1],y)
```

Output:

$$\frac{y-1}{2}$$

Warning.

An attempted function definition such as `f:=lagrange([1,2],[3,4],y)` does not return a function but an expression with respect to y . To define f as a function, input:

```
f:=unapply(lagrange([1,2],[3,4],x),x)
```

Avoid `f(x):=lagrange([1,2],[3,4],x)` since then the Lagrange polynomial would be computed each time `f` is called (indeed in a function definition, the second member of the assignment is not evaluated). Note also that `g(x):=lagrange([1,2],[3,4])` would not work since the default argument of **lagrange** would be global, hence not the same as the local variable used for the definition of `g`.

6.27.30 Trigonometric interpolation: `triginterp`

The `triginterp` command computes a trigonometric polynomial which interpolates given data.

- `triginterp` takes four arguments:
 - L , a list of numbers.
 - a , a number (the beginning of an interval).
 - b , a number (the end of the interval).
 - x , the name of a variable.

The last three arguments can also be given as $x = a..b$.

- `triginterp(L, a, b, x)` or `triginterp(L, x = a..b)` returns the trigonometric polynomial that interpolates data given in the list L . It is assumed that the list L contains ordinate components of the points with equidistant abscissa components between a and b such that the first element of L corresponds to a and the last element to b .

Example.

For example, y may be a list of experimental measurements of some quantity taken in regular intervals, with the first observation at time $t = a$ and the last observation at time $t = b$. The resulting trigonometric polynomial has period

$$T = \frac{n(b-a)}{n-1},$$

where n is the number of observations ($n = \text{size}(y)$). As a specific example, assume that the following data is obtained by measuring the temperature every three hours:

hour of the day	0	3	6	9	12	15	18	21
temperature (°C)	11	10	17	24	32	26	23	19

Furthermore, assume that an estimate of the temperature at 13:45 is required. To obtain a trigonometric interpolation of the data:

Input:

```
tp:=triginterp([11,10,17,24,32,26,23,19],x=0..21)
```

Output:

$$\begin{aligned} & \frac{81}{4} + \frac{1}{8} (-21\sqrt{2} - 42) \cos\left(\frac{1}{12}\pi x\right) + \\ & \frac{1}{8} (-11\sqrt{2} - 12) \sin\left(\frac{1}{12}\pi x\right) + \frac{3}{4} \cos\left(\frac{1}{6}\pi x\right) - \\ & \frac{7}{4} \sin\left(\frac{1}{6}\pi x\right) + \frac{1}{8} (21\sqrt{2} - 42) \cos\left(\frac{1}{4}\pi x\right) + \\ & \frac{1}{8} (-11\sqrt{2} + 12) \sin\left(\frac{1}{4}\pi x\right) + \frac{\cos\left(\frac{1}{3}\pi x\right)}{2} \end{aligned}$$

Now a temperature at 13:45 hrs can be approximated with the value of `tp` for $x = 13.75$.

Input:

```
tp | x=13.75
```

Output:

```
29.4863181684
```

If one of the input parameters is inexact, the result will be inexact too. For example:

Input:

```
Digits:=3;;
triginterp([11,10,17,24,32,26,23,19],x=0..21.0)
```

Output:

```
20.2 - 8.96 cos(0.262x) - 3.44 sin(0.262x) + 0.75 cos(0.524x) -
1.75 sin(0.524x) - 1.54 cos(0.785x) - 0.445 sin(0.785x) + 0.5 cos(1.05x)
```

6.27.31 Natural splines: `spline`

Definition

Let σ_n be a subdivision of a real interval $[a, b]$:

$$a = x_0, \quad x_1, \quad \dots, \quad x_n = b$$

The function s is a *spline* function of degree l if s is a function from $[a, b]$ to \mathbb{R} such that:

- s has continuous derivatives up to the order $l - 1$,
- on each interval of the subdivision σ_n , s is a polynomial of degree less or equal than l .

Theorem

The set of spline functions of degree l on σ_n is an \mathbb{R} -vector space of dimension $n + l$.

Proof.

Let s be a spline function of degree l on σ_n .

On $[a, x_1]$, s is a polynomial A of degree less or equal to l , hence on $[a, x_1]$, $s = A(x) = a_0 + a_1x + \dots + a_lx^l$ and A is a linear combination of $1, x, \dots, x^l$.

On $[x_1, x_2]$, s is a polynomial B of degree less or equal to l , hence on $[x_1, x_2]$, $s = B(x) = b_0 + b_1x + \dots + b_lx^l$. Since s has continuous derivatives up to order $l - 1$,

$$\forall 0 \leq j \leq l - 1, \quad B^{(j)}(x_1) - A^{(j)}(x_1) = 0$$

therefore $B(x) - A(x) = \alpha_1(x - x_1)^l$, i.e. $B(x) = A(x) + \alpha_1(x - x_1)^l$, for some α_1 . Define the function:

$$q_1(x) = \begin{cases} 0 & \text{on } [a, x_1] \\ (x - x_1)^l & \text{on } [x_1, b] \end{cases}$$

so:

$$s|_{[a, x_2]} = a_0 + a_1x + \dots a_lx^l + \alpha_1q_1(x)$$

On $[x_2, x_3]$, s is a polynomial C of degree less or equal than l , hence on $[x_2, x_3]$, $s = C(x) = c_0 + c_1x + \dots c_lx^l$.

Since s has continuous derivatives up to order $l - 1$:

$$\forall 0 \leq j \leq l - 1, \quad C^{(j)}(x_2) - B^{(j)}(x_2) = 0$$

therefore $C(x) - B(x) = \alpha_2(x - x_2)^n$ or $C(x) = B(x) + \alpha_2(x - x_2)^n$.

Define the function:

$$q_2(x) = \begin{cases} 0 & \text{on } [a, x_2] \\ (x - x_2)^l & \text{on } [x_2, b] \end{cases}$$

Hence: $s|_{[a, x_3]} = a_0 + a_1x + \dots a_lx^l + \alpha_1q_1(x) + \alpha_2q_2(x)$

Continuing, define the functions

$$\forall 1 \leq j \leq n - 1, q_j(x) = \begin{cases} 0 & \text{on } [a, x_j] \\ (x - x_j)^l & \text{on } [x_j, b] \end{cases}$$

Then

$$s|_{[a, b]} = a_0 + a_1x + \dots a_lx^l + \alpha_1q_1(x) + \dots + \alpha_{n-1}q_{n-1}(x)$$

and so s is a linear combination of $n+l$ independent functions $1, x, \dots x^l, q_1, \dots q_{n-1}$.

It follows that the set of all possible s is a real vector space of dimension $n + l$.

Types of spline functions

If you want to interpolate a function f on σ_n by a spline function s of degree l , then s must satisfy $s(x_k) = y_k = f(x_k)$ for all $0 \leq k \leq n$. This gives $n + 1$ conditions, leaving $l - 1$ degrees of freedom. You can therefore add $l - 1$ conditions, these conditions are on the derivatives of s at a and b .

Hermite interpolation, natural interpolation and periodic interpolation are three kinds of interpolation obtained by specifying three kinds of constraints. The uniqueness of the solution of the interpolation problem can be proved for each kind of constraints.

If l is odd ($l = 2m - 1$), there are $2m - 2$ degrees of freedom. The constraints are defined by:

- Hermite interpolation:

$$\forall 1 \leq j \leq m - 1, \quad s^{(j)}(a) = f^{(j)}(a), s^{(j)}(b) = f^{(j)}(b)$$

- Natural interpolation:

$$\forall m \leq j \leq 2m - 2, \quad s^{(j)}(a) = s^{(j)}(b) = 0$$

- periodic interpolation:

$$\forall 1 \leq j \leq 2m - 2, \quad s^{(j)}(a) = s^{(j)}(b)$$

If l is even ($l = 2m$), there are $2m - 1$ degrees of freedom. The constraints are defined by:

- Hermite interpolation:

$$\forall 1 \leq j \leq m - 1, \quad s^{(j)}(a) = f^{(j)}(a), s^{(j)}(b) = f^{(j)}(b)$$

and

$$s^{(m)}(a) = f^{(m)}(a)$$

- Natural interpolation:

$$\forall m \leq j \leq 2m - 2, \quad s^{(j)}(a) = s^{(j)}(b) = 0$$

and

$$s^{(2m-1)}(a) = 0$$

- Periodic interpolation:

$$\forall 1 \leq j \leq 2m - 1, \quad s^{(j)}(a) = s^{(j)}(b)$$

6.27.32 Natural interpolation: spline

The `spline` command finds the natural spline.

- `spline` takes four arguments:
 - L_x , a list of abscissas (in increasing order).
 - L_y , a list of ordinates (the same length as L_x).
 - x , a variable name.
 - l , an integer for the degree.
- `spline(L_x, L_y, x, l)` returns the natural spline function s of degree l , where $s(L_{x,j}) = L_{y,j}$ for $j = 0..\text{length}(L_x)$, as a list of polynomials, each polynomial being valid on the corresponding interval determined by L_x .

Examples.

- Find the natural spline of degree 3, crossing through the points $x_0 = 0, y_0 = 1, x_1 = 1, y_1 = 3$ and $x_2 = 2, y_2 = 0$.
Input:

```
spline([0,1,2],[1,3,0],x,3)
```

Output:

$$\left[-\frac{5}{4}x^3 + \frac{13}{4}x + 1, \frac{5}{4}(x-1)^3 - \frac{15}{4}(x-1)^2 - \frac{x-1}{2} + 3 \right]$$

Where the first polynomial, $-\frac{5}{4}x^3 + \frac{13}{4}x + 1$, is defined on the interval $[0, 1]$ (the first interval defined by the list $[0, 1, 2]$) and the second polynomial $\frac{5}{4}(x-1)^3 - \frac{15}{4}(x-1)^2 - \frac{x-1}{2} + 3$ is defined on the interval $[1, 2]$, the second interval defined by the list $[0, 1, 2]$.

- Find the natural spline of degree 4, crossing through the points $x_0 = 0, y_0 = 1, x_1 = 1, y_1 = 3, x_2 = 2, y_2 = 0$ and $x_3 = 3, y_3 = -1$.

Input:

```
spline([0,1,2,3],[1,3,0,-1],x,4)
```

Output:

$$\left[-\frac{62}{121}x^4 + \frac{304}{121}x + 1, \right. \\ \left. \frac{201}{121}(x-1)^4 - \frac{248}{121}(x-1)^3 - \frac{372}{121}(x-1)^2 + \frac{56}{121}(x-1) + 3, \right. \\ \left. -\frac{139}{121}(x-2)^4 + \frac{556}{121}(x-2)^3 + \frac{90}{121}(x-2)^2 - \frac{628}{121}(x-2) \right]$$

Output is a list of three polynomial functions of x , defined respectively on the intervals $[0, 1]$, $[1, 2]$ and $[2, 3]$.

- Find the natural spline interpolation of \cos on $[0, \pi/2, 3\pi/2]$.

Input:

```
spline([0,pi/2,3*pi/2],cos([0,pi/2,3*pi/2]),x,3)
```

Output:

$$\left[\frac{4x^3}{3\pi^3} - \frac{7x}{3\pi} + 1, \right. \\ \left. -\frac{2(x-\frac{\pi}{2})^3}{3\pi^3} + \frac{2(x-\frac{\pi}{2})^2}{\pi^2} - \frac{4(x-\frac{\pi}{2})}{3\pi} \right]$$

6.27.33 Rational interpolation: thiele

The **thiele** command finds the rational interpolation.

- **thiele** takes two arguments:

- *data*, a matrix with two columns. The first column contains the x coordinates and the second column contains the corresponding y coordinates.

Instead of a single matrix, the data can be given as a vector of x coordinates and a vector of y coordinates (in which case the call to **thiele** has three arguments).

– v , an identifier, number or symbolic expression (default: x).

- `thiele(data,v)` returns $R(v)$ where R is the rational interpolant.

Instead of a single matrix `data`, two vectors $\mathbf{x} = (x_1, x_2, \dots, x_n)$ and $\mathbf{y} = (y_1, y_2, \dots, y_n)$ This method computes Thiele interpolated continued function based on the concept of reciprocal differences.

It is not guaranteed that R is continuous, i.e. it may have singularities in the shortest segment which contains all components of the x coordinates.

Examples.

- *Input:*

```
thiele([[1,3],[2,4],[4,5],[5,8]],x)
```

Output:

$$\frac{19x^2 - 45x - 154}{18x - 78}$$

- *Input:*

```
thiele([1,2,a],[3,4,5],3)
```

Output:

$$\frac{13a - 29}{3a - 7}$$

- In the following example, data is obtained by sampling the function $f(x) = (1 - x^4)e^{1-x^3}$.

Input:

```
data_x:=[-1,-0.75,-0.5,-0.25,0,
0.25,0.5,0.75,1,1.25,1.5,1.75,2];
data_y:=[0.0,2.83341735599,2.88770329586,
2.75030303645,2.71828182846,2.66568510781,
2.24894558809,1.21863761951,0.0,-0.555711613283,
-0.377871362418,-0.107135851128,-0.0136782294833];
thiele(data_x,data_y,x)
```

Output:

$$\begin{aligned} & (-1.55286115659x^6 + 5.87298387514x^5 - 5.4439152812x^4 + 1.68655817708x^3 \\ & - 2.40784868317x^2 - 7.55954205222x + 9.40462512097) / \\ & (x^6 - 1.24295718965x^5 - 1.33526268624x^4 + 4.03629272425x^3 \\ & - 0.885419321x^2 - 2.77913222418x + 3.45976823393) \end{aligned}$$

6.27.34 Rational interpolation without poles: `ratinterp`

The `ratinterp` command computes a family of pole-free rational functions which interpolate given data. Rational interpolation usually gives better results than the classic polynomial interpolation, which may oscillate highly in some cases.

- `ratinterp` takes up to three arguments:
 - Matrix `data` with 2 columns with rows corresponding to points (x_k, y_k) , $k = 0, 1, \dots, n$, or the sequence of lists `data_x` = $[x_0, x_1, \dots, x_n]$ and `data_y` = $[y_0, y_1, \dots, y_n]$, where $a = x_0 < x_1 < \dots < x_n = b$,
 - an identifier `var`, which may also be a number, symbolic expression or list of numbers `a` (optional, by default `x`),
 - an integer `d` such that $0 \leq d \leq n$ (optional, by default $\min\{3, \lfloor n/2 \rfloor\}$).
- `ratinterp(data⟨, var, d⟩)` or `ratinterp(data_x, data_y⟨, var, d⟩)` returns a rational interpolation $r(a)$ of the given points using the method of Floater and Hormann (2006). If `a` is a list of numbers a_1, a_2, \dots, a_m , then the list $[r(a_1), \dots, r(a_m)]$ is returned. There are at most $n+1$ distinct interpolants which can be specified by varying the parameter `d`.

Examples.

- Input:

```
ratinterp([1,3,5,8],[2,-1,3,4],x,1)
```

Output:

$$\frac{-19x^3 + 288x^2 - 1133x + 1200}{6x^2 - 48x + 210}$$

- Input:

```
ratinterp([1,3,5,8],[2,-1,3,4],x,2)
```

Output:

$$-\frac{29x^3}{168} + \frac{17x^2}{7} - \frac{1507x}{168} + \frac{61}{7}$$

6.28 Arithmetic and polynomials

Polynomials are represented by expressions or by lists of coefficients in decreasing power order. In the first case, for instructions requiring a main variable (like extended gcd computations), the variable used by default is `x` if not specified. For coefficients in $\mathbb{Z}/n\mathbb{Z}$, use `% n` for each coefficient of the list or apply it to the entire expression defining the polynomial.

6.28.1 The divisors of a polynomial: `divis`

The `divis` command finds the divisors of a polynomial.

- `divis` takes one argument:
 P , a polynomial or a list of polynomials.
- `divis(P)` and returns the list of the divisors of P .

Examples.

- *Input:*

```
divis(x^4-1)
```

Output:

$$\begin{aligned} &[1, x-1, x+1, (x-1)(x+1), x^2+1, (x-1)(x^2+1), \\ &(x+1)(x^2+1), (x-1)(x+1)(x^2+1)] \end{aligned}$$

- *Input:*

```
divis([x^2, x^2-1])
```

Output:

$$[[1, x, x^2], [1, x-1, x+1, (x-1)(x+1)]]$$

6.28.2 Euclidean quotient: `quo` Quo

The `quo` command finds the quotient of the Euclidean division of two polynomials.

- `quo` takes two mandatory arguments and one optional argument:
 - P and Q , two polynomials.
 - Optionally x , the variable (by default x), if P and Q are given as expressions.
- `quo(P, Q, x)` returns the Euclidean quotient of P divided by Q .

Examples.

- *Input:*

```
quo(x^2+2*x+1, x)
```

Output:

$$x+2$$

- *Input:*

```
quo(y^2+2*y+1, y, y)
```

Output:

$$y + 2$$

- In list representation, to get the quotient of $x^2 + 2x + 4$ by $x^2 + x + 2$ you can also input:

$$\text{quo}([1, 2, 4], [1, 1, 2])$$

Output:

$$[1]$$

that is to say, the polynomial 1.

`Quo` is the inert form of `quo`; namely, it evaluates to `quo` for later evaluation. It is used when `Xcas` is in Maple mode (see Section 3.5.2 p.71) to compute the euclidean quotient of the division of two polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$ using Maple-like syntax.

Examples.

- *Input (in Xcas mode):*

$$\text{Quo}(x^2 + 2x + 1, x)$$

Output:

$$\text{quo}(x^2 + 2x + 1, x)$$

- *Input (in Maple mode):*

$$\text{Quo}(x^3 + 3x, 2x^2 + 6x + 5) \bmod 5$$

Output:

$$-2x + 1$$

This division was done using modular arithmetic, unlike with

$$\text{quo}(x^3 + 3x, 2x^2 + 6x + 5) \bmod 5$$

where the division is done in $\mathbb{Z}[X]$ and reduced after to:

$$3x + 6$$

If `Xcas` is not in Maple mode, polynomial division in $\mathbb{Z}/p\mathbb{Z}[X]$ is done e.g. by:

$$\text{quo}((x^3 + 3x) \% 5, (2x^2 + 6x + 5) \% 5)$$

6.28.3 Euclidean remainder: `rem` `Rem`

The `rem` command finds the remainder of the Euclidean division of two polynomials.

- `rem` takes two mandatory arguments and one optional argument:
 - P and Q , two polynomials.
 - Optionally x , the variable (by default x), if P and Q are given as expressions.
- `rem(P, Q, x)` returns the Euclidean remainder of P divided by Q .

Examples.

- *Input:*

```
rem(x^3-1,x^2-1)
```

Output:

$$x - 1$$

- To have the remainder of $x^2 + 2x + 4$ by $x^2 + x + 2$, you can also do:

Input:

```
rem([1,2,4],[1,1,2])
```

Output:

$$[1, 2]$$

i.e. the polynomial $x + 2$.

`Rem` is the inert form of `rem`; namely, it evaluates to `rem` for later evaluation. It is used when `Xcas` is in Maple mode (see Section 3.5.2 p.71) to compute the euclidean remainder of the division of two polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$ using Maple-like syntax.

Examples.

- *Input (in Xcas mode):*

```
Rem(x^3-1,x^2-1)
```

Output:

$$\text{rem}(x^3 - 1, x^2 - 1)$$

- *Input (in Maple mode):*

```
Rem(x^3+3*x,2*x^2+6*x+5) mod 5
```

Output:

$$2x$$

This division was done using modular arithmetic, unlike with

$$\text{rem}(x^3+3x, 2x^2+6x+5) \bmod 5$$

where the division is done in $\mathbb{Z}[X]$ and reduced after to:

$$12x$$

If **Xcas** is not in Maple mode, polynomial division in $\mathbb{Z}/p\mathbb{Z}[X]$ is entered, for example, by:

$$\text{rem}((x^3+3x)\%5, (2x^2+6x+5)\%5)$$

6.28.4 Quotient and remainder: **quorem divide**

The **quorem** command finds the quotient and remainder of the Euclidean division of two polynomials.

divide is a synonym for **quorem**.

- **quorem** takes two mandatory arguments and one optional argument:
 - P and Q , two polynomials.
 - Optionally x , the variable (by default x), if P and Q are given as expressions.
- **quorem**($P, Q \langle, x \rangle$) returns a list consisting of the Euclidean quotient and the Euclidean remainder of P divided by Q .

Examples.

- *Input:*

$$\text{quorem}([1, 2, 4], [1, 1, 2])$$

Output:

$$[[1], [1, 2]]$$

- *Input:*

$$\text{quorem}(x^3-1, x^2-1, x)$$

Output:

$$[x, x-1]$$

6.28.5 GCD of two polynomials with the Euclidean algorithm: **gcd Gcd**

The **gcd** command computes the gcd (greatest common divisor) of polynomials. (See also [6.5.1](#) for GCD of integers.)

- **gcd** takes an unspecified number of arguments:
 - polys*, a sequence or list of polynomials.

- `gcd(polys)` returns the greatest common divisor of the polynomials in *polys*.

Examples.

- *Input:*

$$\text{gcd}(x^2+2x+1, x^2-1)$$
Output:

$$x + 1$$

- *Input:*

$$\text{gcd}(x^2-2x+1, x^3-1, x^2-1, x^2+x-2)$$
or:

$$\text{gcd}([x^2-2x+1, x^3-1, x^2-1, x^2+x-2])$$
Output:

$$x - 1$$

- For polynomials with modular coefficients:

Input (e.g.):

$$\text{gcd}((x^2+2x+1) \bmod 5, (x^2-1) \bmod 5)$$
Output:

$$(1 \% 5)x + 1 \% 5$$
6.28.6 GCD of two polynomials with the Euclidean algorithm: Gcd

`Gcd` is the inert form of `gcd`; namely, it evaluates to `gcd` for later evaluation. It is used when `Xcas` is in Maple mode (see Section 3.5.2 p.71) to compute the gcd of polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$ using Maple-like syntax.

Examples.

- *Input (in Xcas mode):*

$$\text{Gcd}(x^3-1, x^2-1)$$
Output:

$$\text{gcd}(x^3 - 1, x^2 - 1)$$

- *Input (in Maple mode):*

$$\text{Gcd}(x^2+2x, x^2+6x+5) \bmod 5$$
Output:

$$1$$

6.28.7 Choosing the GCD algorithm of two polynomials: ezgcd heugcd modgcd psrgcd

The `ezgcd`, `heugcd`, `modgcd` and `psrgcd` commands compute the gcd (greatest common divisor) of two univariate or multivariate polynomials with coefficients in \mathbb{Z} or $\mathbb{Z}[i]$ with different algorithms.

- `ezgcd`, `heugcd`, `modgcd` and `psrgcd` take two arguments: P and Q , two polynomials.
- `ezgcd(P, Q)` returns the gcd of P and Q computed with the `ezgcd` algorithm.
- `heugcd(P, Q)` returns the gcd of P and Q computed with the heuristic algorithm.
- `modgcd(P, Q)` returns the gcd P and Q computed with the modular algorithm.
- `psrgcd(P, Q)` returns the gcd of P and Q computed with the sub-resultant algorithm.

Examples.

- *Input:*

$$\text{ezgcd}(x^2 - 2xy + y^2 - 1, x - y)$$

or:

$$\text{heugcd}(x^2 - 2xy + y^2 - 1, x - y)$$

or:

$$\text{modgcd}(x^2 - 2xy + y^2 - 1, x - y)$$

or:

$$\text{psrgcd}(x^2 - 2xy + y^2 - 1, x - y)$$

Output:

1

- *Input:*

$$\text{ezgcd}((x+y-1)(x+y+1), (x+y+1)^2)$$

or:

$$\text{heugcd}((x+y-1)(x+y+1), (x+y+1)^2)$$

or:

```
modgcd((x+y-1)*(x+y+1),(x+y+1)^2)
```

Output:

$$x + y + 1$$

- *Input:*

```
psrgcd((x+y-1)*(x+y+1),(x+y+1)^2)
```

Output:

$$-x - y - 1$$

- *Input:*

```
ezgcd((x+1)^4-y^4,(x+1-y)^2)
```

Output:

```
"GCD not successful Error: Bad Argument Value"
```

But:

input:

```
heugcd((x+1)^4-y^4,(x+1-y)^2)
```

or:

```
modgcd((x+1)^4-y^4,(x+1-y)^2)
```

or:

```
psrgcd((x+1)^4-y^4,(x+1-y)^2)
```

Output:

$$x - y + 1$$

6.28.8 LCM of two polynomials: lcm

The `lcm` command computes the LCM (Least Common Multiple) of polynomials. (See 6.5.3 for LCM of integers).

- `lcm` takes an unspecified number of arguments:
polys, a sequence or list of polynomials.
- `lcm(polys)` returns the least common multiple of the polynomials in *polys*.

Examples.

- *Input:*

$$\text{lcm}(x^2+2x+1, x^2-1)$$

Output:

$$(x+1)(x^2-1)$$

• *Input:*

$$\text{lcm}(x, x^2+2x+1, x^2-1)$$

or:

$$\text{lcm}([x, x^2+2x+1, x^2-1])$$

Output:

$$(x^2+x)(x^2-1)$$

6.28.9 Bézout's Identity: `egcd` `gcdex`

Bézout's Identity (also known as Extended Greatest Common Divisor) states that for two polynomials $A(x), B(x)$ with greatest common divisor $D(x)$, there exist polynomials $U(x)$ and $V(x)$ such that

$$U(x) * A(x) + V(x) * B(x) = D(x)$$

The `egcd` computes the greatest common divisor of two polynomials as well as the polynomials $U(x)$ and $V(x)$ in the above identity.

`gcdex` is a synonym for `egcd`.

- `egcd` takes two mandatory arguments and one optional argument:
 - A and B , polynomials given as expressions or lists of coefficients in decreasing order.
 - Optionally, if the polynomials are expressions, x , the variable (which defaults to x).
- `egcd(A, B, x)` returns a list $[U, V, D]$, where D is the greatest common divisor of A and B , and U and V are the polynomials from Bézout's identity.

Examples.

• *Input:*

$$\text{egcd}(x^2+2x+1, x^2-1)$$

Output:

$$[1, -1, 2x+2]$$

• *Input:*

$$\text{egcd}([1, 2, 1], [1, 0, -1])$$

Output:

$$[[1], [-1], [2, 2]]$$

• *Input:*

$$\text{egcd}(y^2-2*y+1, y^2-y+2, y)$$

Output:

$$[y-2, -y+3, 4]$$

• *Input:*

$$\text{egcd}([1, -2, 1], [1, -1, 2])$$

Output:

$$[[1, -2], [-1, 3], [4]]$$

6.28.10 Solving $au+bv=c$ over polynomials: `abcuv`

A consequence of Bézout's identity is that given polynomials $A(x)$, $B(x)$ and $C(x)$, there exist polynomials $U(x)$ and $V(x)$ such that

$$C(x) = U(x) \cdot A(x) + V(x) \cdot B(x)$$

exactly when $C(x)$ is a multiple of the greatest common divisor of $A(x)$ and $B(x)$. The `abcuv` command solves this polynomial equation.

- `abcuv` takes three mandatory and one optional argument:
 - A , B and C , three polynomials given as expressions or lists of coefficients in decreasing order, where C is a multiple of the greatest common divisor of A and B .
 - Optionally if the polynomials are expressions, x , the variable (which defaults to \mathbf{x}).
- `abcuv(A, B, C, x)` returns a list of two expressions $[U, V]$ such that $C = U \cdot A + V \cdot B$.

Examples.

• *Input:*

$$\text{abcuv}(x^2+2*x+1, x^2-1, x+1)$$

Output:

$$\left[\frac{1}{2}, -\frac{1}{2}\right]$$

• *Input:*

$$\text{abcuv}(x^2+2*x+1, x^2-1, x^3+1)$$

Output:

$$\left[\frac{-x+2}{2}, \frac{3}{2}x \right]$$

• *Input:*

$$\text{abcuv}([1, 2, 1], [1, 0, -1], [1, 0, 0, 1])$$

Output:

$$\llbracket \frac{1}{2}, -\frac{1}{2}, \frac{1}{2} \rrbracket, \llbracket -\frac{1}{2}, \frac{1}{2}, -\frac{1}{2} \rrbracket$$

6.28.11 Chinese remainders: chinrem

The Chinese Remainder Theorem states that if $R(x)$ and $Q(x)$ are relatively prime polynomials, then for any polynomials $A(x)$ and $B(x)$, there exists a polynomial $P(x)$ such that:

$$\begin{aligned} P(x) &= A(x) \pmod{R(x)} \\ P(x) &= B(x) \pmod{Q(x)} \end{aligned}$$

The `chinrem` command finds the polynomial P .

- `chinrem` takes two mandatory arguments and one optional argument:
 - $[A, R]$ and $[B, Q]$, two lists, each consisting of two polynomials given by expressions or lists of coefficients in decreasing order.
 - Optionally, if the polynomials are expressions, x , the main variable (by default \mathbf{x}).
- `chinrem([A, R], [B, Q] [, x])` returns the list $[P, S]$, where P and S are polynomials such that:

$$\begin{aligned} S &= RQ \\ P &= A \pmod{R} \\ P &= B \pmod{Q} \end{aligned}$$

If R and Q are coprime, a solution P always exists and all the solutions are congruent modulo $S = R \cdot Q$.

Examples.

- Solve:

$$\begin{cases} P(x) = x & \pmod{x^2 + 1} \\ P(x) = x - 1 & \pmod{x^2 - 1} \end{cases}$$

Input:

$$\text{chinrem}([1, 0], [1, 0, 1], [1, -1], [1, 0, -1])$$

Output:

$$\left[\left[-\frac{1}{2}, 1, -\frac{1}{2} \right], [1, 0, 0, 0, -1] \right]$$

or:

```
chinrem([x,x^2+1],[x-1,x^2-1])
```

Output:

$$\left[-\frac{x^2}{2} + x - \frac{1}{2}, x^4 - 1\right]$$

hence $P(x) = -\frac{x^2 - 2x + 1}{2} \pmod{x^4 - 1}$

• *Input:*

```
chinrem([[1,2],[1,0,1]],[[1,1],[1,1,1]])
```

Output:

$$[-1, -1, 0, 1], [1, 1, 2, 1, 1]$$

or:

```
chinrem([y+2,y^2+1],[y+1,y^2+y+1],y)
```

Output:

$$[-y^3 - y^2 + 1, y^4 + y^3 + 2y^2 + y + 1]$$

6.28.12 Cyclotomic polynomial: cyclotomic

For a positive integer n , *cyclotomic polynomial* of index n is the monic polynomial whose roots are exactly the primitive n th roots of unity (an n th root of unity is primitive if the set of its powers is the set of all the n th roots of unity). Note that this will divide $x^n - 1$, whose roots are all the n th roots of unity.

The `cyclotomic` command computes cyclotomic polynomials.

- `cyclotomic` takes one argument:
 n , an integer.
- `cyclotomic(n)` returns the list of the coefficients of the cyclotomic polynomial of index n .

Examples.

- Let $n = 4$; the fourth roots of unity are: $\{1, i, -1, -i\}$ and the primitive roots are: $\{i, -i\}$. Hence, the cyclotomic polynomial of index 4 is $(x - i)(x + i) = x^2 + 1$.
Input (for verification):

```
cyclotomic(4)
```

Output:

$$[1, 0, 1]$$

- *Input:*

`cyclotomic(5)`

Output:

`[1, 1, 1, 1, 1]`

Hence, the cyclotomic polynomial of index 5 is $x^4 + x^3 + x^2 + x + 1$, which divides $x^5 - 1$ since $(x - 1) * (x^4 + x^3 + x^2 + x + 1) = x^5 - 1$.

- *Input:*

`cyclotomic(10)`

Output:

`[1, -1, 1, -1, 1]`

Hence, the cyclotomic polynomial of index 10 is $x^4 - x^3 + x^2 - x + 1$ and

$$(x^5 - 1) * (x + 1) * (x^4 - x^3 + x^2 - x + 1) = x^{10} - 1$$

- *Input:*

`cyclotomic(20)`

Output:

`[1, 0, -1, 0, 1, 0, -1, 0, 1]`

Hence, the cyclotomic polynomial of index 20 is $x^8 - x^6 + x^4 - x^2 + 1$ and

$$(x^{10} - 1)(x^2 + 1) * (x^8 - x^6 + x^4 - x^2 + 1) = x^{20} - 1$$

6.28.13 Sturm sequences and number of sign changes of P on $(a, b]$: `sturm sturmseq sturmab`

Given a polynomial or rational expression $P(x)$, the Sturm sequence is the sequence $P_1(x), P_2(x), \dots$ given by the recurrence relation:

- $P_1(x)$ is the opposite of the euclidean division remainder of $P(x)$ by $P'(x)$.
- $P_2(x)$ is the opposite of the euclidean division remainder of $P'(x)$ by $P_1(x)$.
- ...

If $P(x)$ is a polynomial of degree n , then this sequence has at most n terms.

If $P(x)$ is square-free, then Sturm's Theorem gives a way to use the sequence to determine the number of zeros of $P(x)$ on an interval.

The `sturm` command can find either the Sturm sequence (in which case it can also be called as `sturmseq`) or the number of zeros in an interval (in which case it can also be called as `sturmab`).

To find the Sturm sequence:

- `sturm` (or `sturmseq`) takes one mandatory argument and one optional argument:
 - P , a polynomial or rational expression.
 - Optionally, x , a variable name (by default x).
- `sturm($P \langle x \rangle$)` (or `sturmseq($P \langle x \rangle$)`) returns the list of the Sturm sequences and multiplicities of the square-free factors of P .

Examples.

- *Input:*

```
sturm(2*x^3+2)
```

or:

```
sturm(2*y^3+2,y)
```

Output:

```
[2, [[1, 0, 0, 1], [3, 0, 0], -9], 1]
```

The first term gives the content of the numerator (here 2), then the Sturm sequence (in list representation) $[x^3 + 1, 3x^2, -9]$.

- *Input:*

```
sturm((2*x^3+2)/(3*x^2+2),x)
```

or:

```
sturmseq((2*x^3+2)/(3*x^2+2),x)
```

Output:

```
[2, [[1, 0, 0, 1], [3, 0, 0], -9], 1, [[3, 0, 2], [6, 0], -72]]
```

The first term gives the content of the numerator (here 2), then the Sturm sequence of the numerator ($[[1,0,0,1],[3,0,0],-9]$), then the content of the denominator (here 1) and the Sturm sequence of the denominator ($[[3,0,2],[6,0],-72]$). As expressions, $[x^3 + 1, 3x^2, -9]$ is the Sturm sequence of the numerator and $[3x^2 + 2, 6x, -72]$ is the Sturm sequence of the denominator.

- *Input:*

```
sturm((x^3+1)^2,x)
```

or:

```
sturmseq((x^3+1)^2,x)
```

Output:

$[1, 1]$

- *Input:*

`sturm(3*(3*x^3+1)/(2*x+2), x)`

Output:

$[3, [[3, 0, 0, 1], [9, 0, 0], -81], 2, [[1, 1], 1]]$

The first term gives the content of the numerator (here 3),
 the second term gives the Sturm sequence of the numerator (here $3x^3+1, 9x^2, -81$),
 the third term gives the content of the denominator (here 2),
 the fourth term gives the Sturm sequence of the denominator ($x+1, 1$).

- *Input:*

`sturm(2*x^3+2, x)`

or:

`sturmseq(2*x^3+2, x)`

Output:

$[2, [[1, 0, 0, 1], [3, 0, 0], -9], 1]$

- *Input:*

`sturm((2*x^3+2)/(x+2), x)`

or:

`sturmseq((2*x^3+2)/(x+2), x)`

Output:

$[2, [[1, 0, 0, 1], [3, 0, 0], -9], 1, [[1, 2], 1]]$

To compute the number of zeros in an interval:

- **sturm** (or **sturmab**) takes four arguments:
 - P , a polynomial expression.
 - x , a variable name.
 - a and b , two real or complex numbers.
- If a and b are reals, **sturm**(P, x, a, b) (or **sturmab**(P, x, a, b)) returns the number of sign changes of P on $(a, b]$; In other words, it returns the number of zeros in $[a, b)$ of the polynomial P/G where $G = \gcd(P, P')$.

- if a or b is complex, `sturm(P, x, a, b)` (or `sturmab(P, x, a, b)`) returns the number of complex roots of P in the rectangle having a and b as opposite vertices.

Examples.

- *Input:*

```
sturm(x^2*(x^3+2),x,-2,0)
```

or:

```
sturmab(x^2*(x^3+2),x,-2,0)
```

Output:

1

- *Input:*

```
sturm(x^2*(x^3+2),x,-2,0)
```

or:

```
sturmab(x^2*(x^3+2),x,-2,0)
```

Output:

1

- *Input:*

```
sturm(x^3-1,x,-2-i,5+3i)
```

or:

```
sturmab(x^3-1,x,-2-i,5+3i)
```

Output:

3

- *Input:*

```
sturm(x^3-1,x,-i,5+3i)
```

Input:

```
sturmab(x^3-1,x,-i,5+3i)
```

Output:

1

Warning!!!!

The polynomial is defined by its symbolic expression.

Input:

```
sturm([1,0,0,1],x)
```

or:

```
sturm([1,0,0,2,0,0],x,-2,0)
```

Output:

```
Bad argument type
```

6.28.14 Sylvester matrix of two polynomials and resultant: sylvester resultant

Given two polynomials $A(x) = \sum_{i=0}^n a_i x^i$ and $B(x) = \sum_{i=0}^m b_i x^i$, their Sylvester matrix is a square matrix of size $m+n$ where $m = \text{degree}(B(x))$ and $n = \text{degree}(A(x))$. The m first lines are made with the $A(x)$ coefficients, so that:

$$\begin{pmatrix} s_{11} = a_n & s_{12} = a_{n-1} & \cdots & s_{1(n+1)} = a_0 & 0 & \cdots & 0 \\ s_{21} = 0 & s_{22} = a_n & \cdots & s_{2(n+1)} = a_1 & s_{2(n+2)} = a_0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ s_{m1} = 0 & s_{m2} = 0 & \cdots & s_{m(n+1)} = a_{m-1} & s_{m(n+2)} = a_{m-2} & \cdots & a_0 \end{pmatrix}$$

and the n further lines are made with the $B(x)$ coefficients, so that:

$$\begin{pmatrix} s_{(m+1)1} = b_m & s_{(m+1)2} = b_{m-1} & \cdots & s_{(m+1)(m+1)} = b_0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ s_{(m+n)1} = 0 & s_{(m+n)2} = 0 & \cdots & s_{(m+n)(m+1)} = b_{n-1} & b_{n-2} & \cdots & b_0 \end{pmatrix}$$

The determinant of a Sylvester polynomial is the resultant of the two polynomials. If A and B have integer coefficients with non-zero resultant r , then the polynomials equation

$$AU + BV = r$$

has a unique solution U, V such that $\text{degree}(U) < \text{degree}(B)$ and $\text{degree}(V) < \text{degree}(A)$, and this solution has integer coefficients.

Remark.

The discriminant of a polynomial is the resultant of the polynomial and its derivative.

The **sylvester** command computes Sylvester matrices.

- **sylvester** takes two arguments:
 P and Q , two polynomials.
- **sylvester**(P, Q) returns the Sylvester matrix of P and Q .

The **resultant** command computes the resultant of two polynomials.

- **resultant** takes three arguments:
 - P and Q , two polynomials.
 - x , a variable.
- **resultant**(P, Q, x) returns the resultant of P and Q .

Example.

Input:

`sylvester(x^3-p*x+q,3*x^2-p,x)`

Output:

$$\begin{bmatrix} 1 & 0 & -p & q & 0 \\ 0 & 1 & 0 & -p & q \\ 3 & 0 & -p & 0 & 0 \\ 0 & 3 & 0 & -p & 0 \\ 0 & 0 & 3 & 0 & -p \end{bmatrix}$$

Input:

`det([[1,0,-p,q,0],[0,1,0,-p,q],[3,0,-p,0,0],
[0,3,0,-p,0],[0,0,3,0,-p]])`

Output:

$$-4p^3 + 27q^2$$

Input:

`resultant(x^3-p*x+q,3*x^2-p,x)`

Output:

$$-4p^3 + 27q^2$$

Examples using the resultant.

- Let $F1$ and $F2$ be two fixed points in the plane and A be a variable point on the circle with center $F1$ and radius $2a$. Find the cartesian equation of the set of points M , intersection of the line $F1A$ and of the perpendicular bisector of $F2A$.

Geometric answer: Since

$$MF1 + MF2 = MF1 + MA = F1A = 2a$$

M is on an ellipse with focus $F1, F2$ and major axis $2a$.

Analytic answer: In the Cartesian coordinate system with center $F1$ and x -axis having the same direction as the vector $F1F2$, the coordinates of A are:

$$A = (2a \cos(\theta), 2a \sin(\theta))$$

where θ is the (Ox, OA) angle. Now choose $t = \tan(\theta/2)$ as parameter, so that the coordinates of A are rational functions with respect to t . More precisely:

$$A = (ax, ay) = (2a \frac{1-t^2}{1+t^2}, 2a \frac{2t}{1+t^2})$$

If $F1F2 = 2c$ and if I is the midpoint of $AF2$, then since the coordinates of $F2$ are $F2 = (2c, 0)$, the coordinates of I are

$$I = (c + ax/2; ay/2) = (c + a \frac{1-t^2}{1+t^2}; a \frac{2t}{1+t^2})$$

IM is orthogonal to $AF2$, hence $M = (x; y)$ satisfies the equation $eq1 = 0$ where

$$eq1 := (x - ix) * (ax - 2 * c) + (y - iy) * ay$$

But $M = (x, y)$ is also on $F1A$, hence M satisfies the equation $eq2 = 0$ where

$$eq2 := y/x - ay/ax$$

The resultant of both equations with respect to t , **resultant**(eq1, eq2, t), is a polynomial $eq3$ depending on the variables x, y , independent of t which is the cartesian equation of the set of points M when t varies.

Input:

```
ax:=2*a*(1-t^2)/(1+t^2); ay:=2*a*2*t/(1+t^2);
ix:=(ax+2*c)/2; iy:=(ay/2)
eq1:=(x-ix)*(ax-2*c)+(y-iy)*ay
eq2:=y/x-ay/ax
factor(resultant(eq1,eq2,t))
```

Output gives as resultant:

$$-(64 \cdot (x^2 + y^2) \cdot (x^2 \cdot a^2 - x^2 \cdot c^2 + -2 \cdot x \cdot a^2 \cdot c + 2 \cdot x \cdot c^3 - a^4 + 2 \cdot a^2 \cdot c^2 + a^2 \cdot y^2 - c^4))$$

The factor $-64 \cdot (x^2 + y^2)$ is always different from zero, hence the locus equation of M :

$$x^2 a^2 - x^2 c^2 + -2 x a^2 c + 2 x c^3 - a^4 + 2 a^2 c^2 + a^2 y^2 - c^4 = 0$$

If the frame origin is O , the middle point of $F1F2$, then this is the cartesian equation of an ellipse. To make the change of origin $\overrightarrow{F1M} = \overrightarrow{F1O} + \overrightarrow{OM}$:

Input:

```
normal(subst(x^2*a^2-x^2*c^2+-2*x*a^2*c+2*x*c^3-a^4+
2*a^2*c^2+ a^2*y^2-c^4,[x,y]=[c+X,Y]))
```

Output:

$$X^2 a^2 - X^2 c^2 + Y^2 a^2 - a^4 + a^2 c^2$$

or if $b^2 = a^2 - c^2$:

Input:

```
normal(subst(-c^2*X^2+c^2*a^2+X^2*a^2-a^4+a^2*Y^2,c^2=a^2-b^2))
```

Output:

$$X^2b^2 + Y^2a^2 - a^2b^2$$

that is to say, after division by $a^2 * b^2$, M satisfies the equation:

$$\frac{X^2}{a^2} + \frac{Y^2}{b^2} = 1$$

- Let $F1$ and $F2$ be fixed points and A a variable point on the circle with center $F1$ and radius $2a$. Find the cartesian equation of the hull of D , the segment bisector of $F2A$.

The segment bisector of $F2A$ is tangent to the ellipse of focus $F1, F2$ and major axis $2a$.

In the Cartesian coordinate system with center $F1$ and x -axis having the same direction as the vector $F1F2$, the coordinates of A are:

$$A = (2a \cos(\theta); 2a \sin(\theta))$$

where θ is the (Ox, OA) angle. Choose $t = \tan(\theta/2)$ as parameter such that the coordinates of A are rational functions with respect to t . More precisely:

$$A = (ax; ay) = (2a \frac{1-t^2}{1+t^2}; 2a \frac{2t}{1+t^2})$$

If $F1F2 = 2c$ and I is the midpoint of $AF2$:

$$F2 = (2c, 0), \quad I = (c + ax/2; ay/2) = (c + a \frac{1-t^2}{1+t^2}; a \frac{2t}{1+t^2})$$

Since D is orthogonal to $AF2$, the equation of D is $eq1 = 0$ where

$$eq1 := (x - ix) * (ax - 2 * c) + (y - iy) * ay$$

So, the hull of D is the locus of M , the intersection point of D and D' where D' has equation $eq2 := \text{diff}(eq1, t) = 0$.

Input:

```
ax:=2*a*(1-t^2)/(1+t^2); ay:=2*a*2*t/(1+t^2);
ix:=(ax+2*c)/2; iy:=(ay/2)
eq1:=normal((x-ix)*(ax-2*c)+(y-iy)*ay)
eq2:=normal(diff(eq1,t))
factor(resultant(eq1,eq2,t))
```

Output gives as resultant:

$$(-(64a^2))(x^2 + y^2)(x^2a^2 - x^2c^2 - 2xa^2c + 2xc^3 - a^4 + 2a^2c^2 + a^2y^2 - c^4)$$

The factor $-64 \cdot a^2 \cdot (x^2 + y^2)$ is always different from zero, therefore the locus equation is:

$$x^2a^2 - x^2c^2 - 2xa^2c + 2xc^3 - a^4 + 2a^2c^2 + a^2y^2 - c^4 = 0$$

If O , the midpoint of $F1F2$, is chosen as origin, you find again the cartesian equation of the ellipse:

$$\frac{X^2}{a^2} + \frac{Y^2}{b^2} = 1$$

6.29 Exact bounds for roots of a polynomial

6.29.1 Exact bounds for real roots of a polynomial: `realroot`

The `realroot` command finds bounds for the real roots of a polynomial.

- `realroot` takes two mandatory arguments and two optional arguments:
 - P , a polynomial.
 - ϵ , a positive real number.
 - Optionally, a , b , two complex numbers.
- `realroot(P, ϵ)` returns a list of vectors, where the elements of each vector are a list containing one of:
 - an interval of length less than ϵ containing a real root of the polynomial and the multiplicity of this root.
 - the value of an exact real root of the polynomial and the multiplicity of this root.
- `realroot(P, ϵ, a, b)` returns a list of vectors as above, but only for the roots lying in the interval $[a, b]$.

Examples.

- Find the real roots of $x^3 + 1$.

Input:

```
realroot(x^3+1, 0.1)
```

Output:

```
[ -1  1 ]
```

- Find the real roots of $x^3 - x^2 - 2x + 2$.

Input:

```
realroot(x^3-x^2-2*x+2, 0.1)
```

Output:

```
[ -[1.40624999999999..1.500000000000001]  1 ]
[ 1                                     1 ]
[ 1.37499999999999..1.437500000000001]  1 ]
```

- Find the real roots of $x^3 - x^2 - 2x + 2$ in the interval $[0; 2]$.

Input:

```
realroot(x^3-x^2-2*x+2, 0.1,0,2)
```

Output:

```
[ 1                                     1 ]
[ 1.37499999999999..1.437500000000001]  1 ]
```

6.29.2 Exact bounds for complex roots of a polynomial: `complexroot`

The `complexroot` command finds bounds for the complex roots of a polynomial.

- `complexroot` takes two mandatory arguments and two optional arguments:
 - P , a polynomial.
 - ϵ , a positive real number.
 - Optionally, α, β , two complex numbers.
- `complexroot(P, ϵ)` returns a list of vectors, where the elements of each vector are one of:
 - an interval (the boundaries of this interval are the opposite vertices of a rectangle with sides parallel to the axis and containing a complex root of the polynomial) and the multiplicity of this root. Suppose the interval is $[a_1 + ib_1, a_2 + ib_2]$ then $|a_1 - a_2| < \epsilon$, $|b_1 - b_2| < \epsilon$ and the root $a + ib$ satisfies $a_1 \leq a \leq a_2$ and $b_1 \leq b \leq b_2$.
 - the value of an exact complex root of the polynomial and the multiplicity of this root.
- `complexroot($P, \epsilon, \alpha, \beta$)` returns a list of vectors as above, but only for the roots lying in the rectangle with sides parallel to the axis having α, β as opposite vertices.

Examples.

- Find the roots of $x^3 + 1$.

Input:

```
complexroot(x^3+1,0.1)
```

Output:

$$\begin{bmatrix} -1 \\ [0.499999046325680..0.500000953674320] - [0.866024494171135..0.866026401519779]i \\ [0.499999046325680..0.500000953674320] + [0.866024494171135..0.866026401519779]i \end{bmatrix}$$

Hence, for $x^3 + 1$:

- -1 is a root of multiplicity 1,
- $a+ib$ is a root of multiplicity 1 with $0.499999046325680 \leq a \leq 0.500000953674320$ and $-0.866026401519779 \leq b \leq -0.866024494171135$.
- $c+id$ is a root of multiplicity 1 with $0.499999046325680 \leq c \leq 0.500000953674320$ and $0.866024494171135 \leq d \leq 0.866026401519779$.

- Find the roots of x^3+1 lying inside the rectangle with opposite vertices $-1, 1+2*i$.

Input:

`complexroot(x^3+1,0.1,-1,1+2*i)`

Output:

$$\begin{bmatrix} -1 & 1 \\ [0.499999046325680..0.500000953674320] + [0.866024494171135..0.866026401519779]i & 1 \end{bmatrix}$$

6.29.3 Exact bounds for real roots of a polynomial: VAS

The `VAS` command uses the Vincent-Akritas-Strzebonski algorithm to find intervals containing the real roots of polynomials.

- `VAS` takes one argument:
 P , a polynomial.
- `VAS(P)` returns a list of intervals which contain the real roots of P , where each interval contains exactly one root.

Examples.

- *Input:*

`VAS(x^3 - 7*x + 7)`

Output:

$$\begin{bmatrix} -4 & 0 \\ 1 & \frac{3}{2} \\ \frac{3}{2} & 2 \end{bmatrix}$$

- *Input:*

`VAS(x^5 + 2*x^4 - 6*x^3 - 7*x^2 + 7*x + 7)`

Output:

$$\left[[-5, -1], -1, \left[1, \frac{3}{2} \right], \left[\frac{3}{2}, 2 \right] \right]$$

- *Input:*

`VAS(x^3 - x^2 - 2*x + 2)`

Output:

$$[[-3, 0], 1, [1, 3]]$$

6.29.4 Exact bounds for positive real roots of a polynomial: `VAS_positive`

The `VAS_positive` command uses the Vincent-Akritas-Strzebonski algorithm to find intervals containing the positive real roots of polynomials.

- `VAS_positive` takes one argument:
 P , a polynomial.
- `VAS_positive(P)` returns a list of intervals which contain the positive real roots of P , where each interval contains exactly one root.

Examples.

- *Input:*

`VAS_positive(x^3 - 7*x + 7)`

Output:

$$\left[\begin{array}{cc} 1 & \frac{3}{2} \\ \frac{3}{2} & 2 \end{array} \right]$$

- *Input:*

`VAS_positive(x^5 + 2*x^4 - 6*x^3 - 7*x^2 + 7*x + 7)`

Output:

$$\left[\begin{array}{cc} 1 & \frac{3}{2} \\ \frac{3}{2} & 2 \end{array} \right]$$

- *Input:*

`VAS_positive(x^3 - x^2 - 2*x + 2)`

Output:

$$[1, [1, 3]]$$

6.29.5 An upper bound for the positive real roots of a polynomial: `posubLMQ`

The `posubLMQ` command uses the Local Max Quadratic (LMQ) Akritas-Strzebonski-Vigklas algorithm to find upper bounds for the positive real roots of polynomials.

- `posubLMQ` takes one argument:
 P , a polynomial.
- `posubLMQ(P)` returns a (non-optimal) upper bound for the positive real roots of P .

Examples.

- *Input:*

`posubLMQ(x^3 - 7*x + 7)`

Output:

4

- *Input:*

`posubLMQ(x^5 + 2*x^4 - 6*x^3 - 7*x^2 + 7*x + 7)`

Output:

4

- *Input:*

`posubLMQ(x^3 - x^2 - 2*x + 2)`

Output:

3

6.29.6 A lower bound for the positive real roots of a polynomial: `poslbdLMQ`

The `poslbdLMQ` command uses the Local Max Quadratic (LMQ) Akritas-Strzebonski-Vigklas algorithm to find lower bounds for the positive real roots of polynomials.

- `poslbdLMQ` takes one argument:
 P , a polynomial.
- `poslbdLMQ(P)` returns a (non-optimal) lower bound for the positive real roots of P .

Examples.

- *Input:*

`poslbdLMQ(x^3 - 7*x + 7)`

Output:

$\frac{1}{2}$

- *Input:*

`poslbdLMQ(x^5 + 2*x^4 - 6*x^3 - 7*x^2 + 7*x + 7)`

Output:

$\frac{1}{2}$

- *Input:*

`poslbdLMQ(x^3 - x^2 - 2*x + 2)`

Output:

$\frac{1}{2}$

6.29.7 Exact values of rational roots of a polynomial: rationalroot

The `rationalroot` command finds rational roots of polynomials.

- `rationalroot` takes one mandatory and two optional arguments:
 - P , a polynomial.
 - Optionally, α and β , two real numbers.
- `rationalroot(P)` returns the list of the value of the rational roots of P without multiplicity.
- `rationalroot(P, α, β)` returns the list of the rational roots of P which are in the interval $[\alpha, \beta]$.

Examples.

- Find the rational roots of $2 * x^3 - 3 * x^2 - 8 * x + 12$:

Input:

```
rationalroot(2*x^3-3*x^2-8*x+12)
```

Output:

$$\left[2, -2, \frac{3}{2} \right]$$

- Find the rational roots of $2 * x^3 - 3 * x^2 - 8 * x + 12$ in $[1, 2]$:

Input:

```
rationalroot(2*x^3-3*x^2-8*x+12,1,2)
```

Output:

$$\left[2, \frac{3}{2} \right]$$

- Find the rational roots of $2 * x^3 - 3 * x^2 + 8 * x - 12$:

Input:

```
rationalroot(2*x^3-3*x^2+8*x-12)
```

Output:

$$\left[\frac{3}{2} \right]$$

- Find the rational roots of $2 * x^3 - 3 * x^2 + 8 * x - 12$:

Input:

```
rationalroot(2*x^3-3*x^2+8*x-12)
```

Output:

$$\left[\frac{3}{2} \right]$$

- Find the rational roots of $(3x-2)^2(2x+1) = 18x^3 - 15x^2 - 4x + 4$.
Input:

```
rationalroot(18*x^3-15*x^2-4*x+4)
```

Output:

$$\left[-\frac{1}{2}, \frac{2}{3} \right]$$

6.29.8 Exact values of the complex rational roots of a polynomial: `crationalroot`

The `crationalroot` command finds complex rational roots of polynomials.

- `crationalroot` takes one mandatory and two optional arguments:
 - P , a polynomial.
 - Optionally, α and β , two complex numbers.
- `crationalroot(P)` returns the list of the value of the rational roots of P without multiplicity.
- `crationalroot(P, α, β)` returns the list of the rational roots of P which are in the rectangle with sides parallel to the axis having $[\alpha, \beta]$ as opposite vertices.

Example.

Find the rational complex roots of $(x^2+4)(2x-3) = 2x^3 - 3x^2 + 8x - 12$:

Input:

```
crationalroot(2*x^3-3*x^2+8*x-12)
```

Output:

$$\left[2i, \frac{3}{2}, -2i \right]$$

6.30 Orthogonal polynomials

6.30.1 Legendre polynomials: `legendre`

The Legendre polynomial $L(n, x)$ of degree n is a polynomial solution of the differential equation

$$(x^2 - 1)y'' - 2xy' - n(n+1)y = 0$$

The Legendre polynomials satisfy the recurrence relation:

$$L(0, x) = 1$$

$$L(1, x) = x$$

$$L(n, x) = \frac{2n-1}{n}xL(n-1, x) - \frac{n-1}{n}L(n-2, x)$$

These polynomials are orthogonal for the scalar product:

$$\langle f, g \rangle = \int_{-1}^{+1} f(x)g(x) dx$$

The **legendre** command finds the Legendre polynomials.

- **legendre** takes one mandatory argument and one optional argument:
 - n , an integer.
 - Optionally, x , a variable name (by default **x**).
- **legendre**($n \langle, x \rangle$) returns the Legendre polynomial of degree n .

Examples.

- *Input:*

legendre(4)

Output:

$$\frac{35}{8}x^4 - \frac{15}{4}x^2 + \frac{3}{8}$$

- *Input:*

legendre(4,y)

Output:

$$\frac{35}{8}y^4 - \frac{15}{4}y^2 + \frac{3}{8}$$

6.30.2 Hermite polynomial: **hermite**

The Hermite polynomials $H(n, x)$ satisfy the recurrence relation:

$$\begin{aligned} H(0, x) &= 1 \\ H(1, x) &= 2x \\ H(n, x) &= 2xH(n-1, x) - 2(n-1)H(n-2, x) \end{aligned}$$

These polynomials are orthogonal for the scalar product:

$$\langle f, g \rangle = \int_{-\infty}^{+\infty} f(x)g(x)e^{-x^2} dx$$

The **hermite** command finds the Hermite polynomials.

- **hermite** takes one mandatory argument and one optional argument:
 - n , an integer.
 - Optionally, x , a variable name (by default **x**).
- **hermite**($n \langle, x \rangle$) returns the Hermite polynomial of degree n .

Examples.

- *Input:*

$$\text{hermite}(6)$$

Output:

$$64x^6 - 480x^4 + 720x^2 - 120$$

- *Input:*

$$\text{hermite}(6, y)$$

Output:

$$64y^6 - 480y^4 + 720y^2 - 120$$

6.30.3 Laguerre polynomials: laguerre

The Laguerre polynomial of degree n and parameter a satisfy the following recurrence relation:

$$L(0, a, x) = 1$$

$$L(1, a, x) = 1 + a - x$$

$$L(n, a, x) = \frac{2n + a - 1 - x}{n} L(n - 1, a, x) - \frac{n + a - 1}{n} L(n - 2, a, x)$$

These polynomials are orthogonal for the scalar product

$$\langle f, g \rangle = \int_0^{+\infty} f(x)g(x)x^a e^{-x} dx$$

The **laguerre** command finds the Laguerre polynomials.

- **laguerre** takes one mandatory argument and two optional arguments:
 - n , an integer.
 - Optionally, x , a variable name (by default **x**).
 - Optionally, a , a parameter name (by default **a**).
- **laguerre**($n \langle, x, a \rangle$) returns the Laguerre polynomial of degree n and parameter a .

Examples.

- *Input:*

$$\text{laguerre}(2)$$

Output:

$$\frac{1}{2}a^2 - ax + \frac{3}{2}a + \frac{1}{2}x^2 - 2x + 1$$

- *Input:*

laguerre(2,y)

Output:

$$\frac{1}{2}a^2 - ay + \frac{3}{2}a + \frac{1}{2}y^2 - 2y + 1$$

- *Input:*

laguerre(2,y,b)

Output:

$$\frac{1}{2}b^2 - by + \frac{3}{2}b + \frac{1}{2}y^2 - 2y + 1$$

6.30.4 Tchebychev polynomials of the first kind: tchebyshev1

The Tchebychev polynomial of first kind $T(n, x)$ is defined by

$$T(n, x) = \cos(n \arccos(x))$$

and satisfy the recurrence relation:

$$T(0, x) = 1, \quad T(1, x) = x, \quad T(n, x) = 2xT(n-1, x) - T(n-2, x)$$

The polynomials $T(n, x)$ are orthogonal for the scalar product

$$\langle f, g \rangle = \int_{-1}^{+1} \frac{f(x)g(x)}{\sqrt{1-x^2}} dx$$

The `tchebyshev1` command finds the Tchebychev polynomials of the first kind.

- `tchebyshev1` takes one mandatory argument and one optional argument:
 - n , an integer.
 - Optionally x , a variable name (by default x).
- `tchebyshev1(n, x)` returns the Tchebychev polynomial of first kind of degree n .

Examples.

- *Input:*

tchebyshev1(4)

Output:

$$8x^4 - 8x^2 + 1$$

- *Input:*

`tchebyshev1(4,y)`

Output:

$$8y^4 - 8y^2 + 1$$

Indeed

$$\begin{aligned}\cos(4x) &= \operatorname{Re}((\cos(x) + i\sin(x))^4) \\ &= \cos(x)^4 - 6\cos(x)^2(1 - \cos(x)^2) + ((1 - \cos(x)^2)^2) \\ &= T(4, \cos(x))\end{aligned}$$

6.30.5 Tchebychev polynomial of the second kind: `tchebyshev2`

The Tchebychev polynomial of second kind $U(n, x)$ is defined by:

$$U(n, x) = \frac{\sin((n+1) \cdot \arccos(x))}{\sin(\arccos(x))}$$

or equivalently:

$$\sin((n+1)x) = \sin(x) * U(n, \cos(x))$$

These satisfy the recurrence relation:

$$\begin{aligned}U(0, x) &= 1 \\ U(1, x) &= 2x \\ U(n, x) &= 2xU(n-1, x) - U(n-2, x)\end{aligned}$$

The polynomials $U(n, x)$ are orthogonal for the scalar product

$$\langle f, g \rangle = \int_{-1}^{+1} f(x)g(x)\sqrt{1-x^2}dx$$

The `tchebyshev2` command finds the Tchebychev polynomials of the first kind.

- `tchebyshev2` takes one mandatory argument and one optional argument:
 - n , an integer.
 - Optionally x , a variable name (by default \mathbf{x}).
- `tchebyshev2(n⟨,x⟩)` returns the Tchebychev polynomial of second kind of degree n .

Examples.

- *Input:*

tchebyshev2(3)

Output:

$$8x^3 - 4x$$

• *Input:*

tchebyshev2(3,y)

Output:

$$8y^3 - 4y$$

Indeed:

$$\sin(4x) = \sin(x) * (8 * \cos(x)^3 - 4 \cos(x)) = \sin(x) * U(3, \cos(x))$$

6.31 Gröbner basis and Gröbner reduction

6.31.1 Gröbner basis: gbasis

A set of polynomials $\{F_1, \dots, F_N\}$ generate an *ideal* I ; namely, I is the set of all linear combinations of the F_j . Given such an ideal, a Gröbner basis for I is a subset $G = \{G_1, \dots, G_n\}$ of I such that for any F in I , there is a G_k in G such that the leading monomial of G_k divides the leading monomial of F . (Note that the leading monomial depends on a fixed ordering of the monomials.)

If G is a Gröbner basis for such an ideal I , then for any nonzero F in I , if you do a Euclidean division of F by the corresponding G_k , take the remainder of this division, do again the same and so on, at some point you get a remainder of zero.

Example.

Let I be the ideal generated by $\{x^3 - 2xy, x^2y - 2y^2 + x\}$ with the standard lexicographic order on the monomials. One Gröbner basis for I is

$$G = \{g_1(x, y) = x^2, g_2(x, y) = xy, g_3(x, y) = 2y^2 - x\}$$

Consider the element $F(x, y) = 2x^2y - 3x^2 + 6xy - 4y^2 + 2x$ of I . The leading monomial x^2y of $F(x, y)$ is divisible by the leading monomial x^2 of $g_1(x, y)$. Dividing $F(x, y)$ by $g_1(x, y)$ leaves a remainder of $R_1(x, y) = 6xy - 4y^2 + 2x$. The leading monomial of $R_1(x, y)$, which is xy , is divisible by the leading monomial of $g_2(x, y)$, which is xy . Dividing $R_1(x, y)$ by $g_2(x, y)$ leaves a remainder of $R_2(x, y) = -4y^2 + 2x$. Finally, the leading monomial of $R_2(x, y)$, which is y^2 , is divisible by the leading monomial of $g_3(x, y)$, which is y^2 . Dividing $R_2(x, y)$ by $g_3(x, y)$ leaves a remainder of 0.

The `gbasis` command computes Gröbner bases.

- `gbasis` takes two mandatory arguments and three optional arguments:
 - *polys*, a list of polynomials.
 - *vars*, a list of the variable names.

- Optionally, *order*, which can be one of:
 - * **plex**, to order the monomials lexicographically (this is the default).
 - * **tdeg**, to order the monomials first by total degree then by lexicographic order.
 - * **revlex**, to order the monomials reverse lexicographically.
 - Optionally, **with_cocoa=boolean**, where *boolean* can be **true** or **false**. A value of **true** means to use the CoCoA library to compute the Gröbner basis, a value of **false** means not to use it. A value of **true** is recommended, but requires that CoCoA support be compiled into Xcas.
 - Optionally, **with_f5=boolean**, where *boolean* can be **true** or **false**. A value of **true** means to use the F5 algorithm of the CoCoA library, a value of **false** means not to use it. If this is **true**, then the polynomials are homogenized and so the specified order is not used.
- **gbasis**(*polys*, *vars* \langle , *order*, **with_cocoa=boolean**, **with_f5=boolean** \rangle) returns a Gröbner basis of the ideal spanned by polynomials in *polys*.

Note that the lexicographic order depends on the order the variables are given in *vars*. For example, if *vars*=[x,y,z], then $x^2y^4z^3$ comes before $x^2y^3z^4$, but if *vars*=[x,z,y], then $x^2y^4z^3$ comes after $x^2y^3z^4$.

Examples.

- *Input:*

```
gbasis([2*x*y-y^2,x^2-2*x*y],[x,y])
```

Output:

$$[y^3, x^2 - y^2, 2xy - y^2]$$

Input:

```
gbasis([x1+x2+x3,x1*x2+x1*x3+x2*x3,x1*x2*x3-1],
[x1,x2,x3],tdeg,with_cocoa=false)
```

Output:

$$[x_3^3 - 1, -x_2^2 - x_2x_3 - x_3^2, x_1 + x_2 + x_3]$$

6.31.2 Gröbner reduction: greduce

The **greduce** command will find a polynomial modulo I , where I is an ideal as in Section 6.31.1 p.408.

- **greduce** takes three arguments mandatory arguments and three optional arguments:
 - P , a multivariate polynomial.

- *gbasis*, a vector made of polynomials which is supposed to be a Gröbner basis.
- *vars*, and a vector of variable names.
- Optionally, the same ordering options and CoCaA options as **gbasis** (see Section 6.31.1 p.408).
- **greduce**(*P*, *gbasis*, *vars* \langle , *options* \rangle) returns the reduction of *P* with respect to the Gröbner basis *gbasis*. It is 0 if and only if the polynomial belongs to the ideal.

Examples.

- *Input:*

```
greduce(x*y-1,[x^2-y^2,2*x*y-y^2,y^3],[x,y])
```

Output:

$$\frac{1}{2}y^2 - 1$$

that is to say $xy - 1 = \frac{1}{2}y^2 - 1 \pmod{I}$ where *I* is the ideal generated by the Gröbner basis $[x^2 - y^2, 2xy - y^2, y^3]$, because $\frac{1}{2}y^2 - 1$ is the Euclidean division remainder of $xy - 1$ by $G_2 = 2xy - y^2$.

- *Input:*

```
greduce(x1^2*x3^2,[x3^3-1,-x2^2-x2*x3-x3^2,x1+x2+x3],
        [x1,x2,x3],tdeg)
```

Output:

$$x_2$$

6.31.3 Testing if a polynomial or list of polynomials belongs to an ideal given by a Gröbner basis: **in_ideal**

The **in_ideal** command determines whether or not a polynomial is in an ideal.

- **in_ideal** takes three mandatory arguments and one optional argument:
 - *P*, a polynomial or a list of polynomials.
 - *gbasis*, a list giving a Gröbner basis.
 - *vars*, the list of polynomial variables.
If *gbasis* is computed with a different order from the default, then *vars* must use the same order.
 - Optionally, an optional argument from **gbasic** (see Section 6.31.1 p.408), such as **plex** or **tdeg**. By default it will be **plex**.

- `in_ideal(P,gbasis,vars <, option >)` returns the value `true` (1) or `false` (0), or a list of `true`s and `false`s, indicating whether or not the polynomial(s) in P are in the ideal generated by $gbasis$ using the variables in $vars$.

Examples.

- *Input:*

```
in_ideal((x+y)^2,[y^2,x^2 + 2*x*y],[x,y])
```

Output:

1

- *Input:*

```
in_ideal([(x+y)^2,x+y],[y^2,x^2+2*x*y],[x,y])
```

Output:

[1, 0]

- *Input:*

```
in_ideal(x+y,[y^2,x^2+2*x*y],[x,y])
```

Output:

0

6.31.4 Building a polynomial from its evaluation: genpoly

The `genpoly` command finds a polynomial which evaluates to a given polynomial.

- `genpoly` takes three arguments:
 - P , a polynomial with $n - 1$ variables.
 - b , an integer.
 - x , the name of a variable.
- `genpoly(P,b,x)` returns the polynomial Q with n variables (the $n - 1$ variables in P and the variable x) such that the coefficients of Q are in the interval $(-b/2, b/2]$ and $Q|_{x=b} = P$. In other words, P is written in base b but using the convention that the Euclidean remainder belongs to $(-b/2, b/2]$ (this convention is also known as s-mod representation).

Examples.

- *Input:*

```
genpoly(61,6,x)
```

Output:

$$2x^2 - 2x + 1$$

Indeed 61 divided by 6 is 10 with remainder 1, then 10 divided by 6 is 2 with remainder -2 (instead of the usual quotient 1 and remainder 4 out of bounds),

$$61 = 2 * 6^2 - 2 * 6 + 1$$

• *Input:*

$$\text{genpoly}(5, 6, x)$$

Output:

$$x - 1$$

Indeed: $5 = 6 - 1$.

• *Input:*

$$\text{genpoly}(7, 6, x)$$

Output:

$$x + 1$$

Indeed: $7 = 6 + 1$

• *Input:*

$$\text{genpoly}(7*y+5, 6, x)$$

Output:

$$xy + x + y - 1$$

Indeed: $x * y + x + y - 1 = y(x + 1) + (x - 1)$.

• *Input:*

$$\text{genpoly}(7*y+5*z^2, 6, x)$$

Output:

$$xy + xz^2 + y - z^2$$

Indeed: $x * y + x * z + y - z = y * (x + 1) + z * (x - 1)$.

6.32 Rational functions

6.32.1 Numerator: `getNum`

The `getNum` command finds the numerator of an unreduced rational function.

- `getNum` takes one argument:
rat, a rational function.
- `getNum(rat)` returns the numerator of *rat*.

Unlike `numer` (see Section 6.32.2 p.413), `getNum` does not simplify the expression before extracting the numerator.

Examples.

- *Input:*

```
getNum((x^2-1)/(x-1))
```

Output:

$$x^2 - 1$$

- *Input:*

```
getNum((x^2+2*x+1)/(x^2-1))
```

Output:

$$x^2 + 2x + 1$$

6.32.2 Numerator after simplification: `numer`

The `numer` command finds the numerator of a rational function, after it has been reduced. (See also 6.7.3.)

- `numer` takes one argument:
rat, a rational function.
- `numer(rat)` returns the numerator of the irreducible representation of *rat*.

Examples.

- *Input:*

```
numer((x^2-1)/(x-1))
```

Output:

$$x + 1$$

- *Input:*

```
numer((x^2+2*x+1)/(x^2-1))
```

Output:

$$x + 1$$

6.32.3 Denominator: `getDenom`

The `getDenom` command finds the denominator of an unreduced rational function.

- `getDenom` takes one argument:
rat, a rational function.
- `getDenom(rat)` returns the denominator of *rat*.

Unlike `denom` (see Section 6.32.4 p.414), `textttgetDenom` does not simplify the expression before extracting the denominator.

Examples.

- *Input:*

```
getDenom((x^2-1)/(x-1))
```

Output:

$$x - 1$$

- *Input:*

```
getDenom((x^2+2*x+1)/(x^2-1))
```

Output:

$$x^2 - 1$$

6.32.4 Denominator after simplification: `denom`

The `denom` command finds the denominator of a rational function, after it has been reduced. (See also 6.7.4.)

- `denom` takes one argument:
rat, a rational function.
- `denom(rat)` returns the denominator of the irreducible representation of *rat*.

Examples.

- *Input:*

```
denom((x^2-1)/(x-1))
```

Output:

$$1$$

- *Input:*

```
denom((x^2+2*x+1)/(x^2-1))
```

Output:

$$x - 1$$

6.32.5 Numerator and denominator: f2nd fxnd

The `f2nd` command finds the numerator and denominator of rational function, after simplification.

`fxnd` is a synonym for `f2nd`.

- `f2nd` takes one argument:
rat, a rational function.
- `f2nd(rat)` returns the list of the numerator and the denominator of the irreducible representation of *rat*.

Examples.

- *Input:*

```
f2nd((x^2-1)/(x-1))
```

Output:

```
[x + 1, 1]
```

- *Input:*

```
f2nd((x^2+2*x+1)/(x^2-1))
```

Output:

```
[x + 1, x - 1]
```

6.32.6 Simplifying: simp2

The `simp2` command removes common factors from a pair of polynomials, as if reducing the numerator and denominator of a rational function. (See also Section 6.7.6 p.165.)

- `simp2` takes two arguments:
P and *Q*, two polynomials (or two integers, see Section 6.7.6 p.165).
- `simp2(P,Q)` returns a list of two polynomials seen as the numerator and denominator of the irreducible representation of the rational function P/Q .

Example.

Input:

```
simp2(x^3-1,x^2-1)
```

Output:

```
[x^2 + x + 1, x + 1]
```

6.32.7 Common denominator: comDenom

The `comDenom` command finds the common denominator of a sum of rational functions and adds them.

- `comDenom` takes one argument:
sum, a sum of rational functions.
- `comDenom(sum)` returns *sum* with the terms combined over a common denominator.

Example.

Input:

```
comDenom(x-1/(x-1)-1/(x^2-1))
```

Output:

$$\frac{x^3 - 2x - 2}{x^2 - 1}$$

6.32.8 Polynomial and fractional part: propfrac

The `propfrac` command rewrites a rational function as a polynomial plus a rational function whose numerator has smaller degree than the denominator; namely, it writes $\frac{A(x)}{B(x)}$ (after reduction), as:

$$Q(x) + \frac{R(x)}{B(x)} \quad \text{where } R(x) = 0 \text{ or } 0 \leq \text{degree}(R(x)) < \text{degree}(B(x))$$

(See also Section [6.7.2](#) p.163.)

- `propfrac` takes one argument:
rat, a rational function.
- `propfrac(rat)` returns the sum of a polynomial and rational function which add to *rat*, and with the degree of the numerator of the rational function less than the degree of the denominator.

Example.

Input:

```
propfrac((5*x+3)*(x-1)/(x+2))
```

Output:

$$5x - 12 + \frac{21}{x + 2}$$

6.32.9 Partial fraction expansion: partfrac cpartfrac

The `partfrac` and `cpartfrac` commands find the partial fraction expansion of a rational function.

- `partfrac` takes one argument:
rat, a rational function.

- `partfrac(rat)` returns the partial fraction expansion of *rat*.
The `partfrac` command is equivalent to the `convert` command (see Section 6.23.26 p.323) with `parfrac` (or `partfrac` or `fullparfrac`) as option.
- `cpartfrac(rat)` behaves just like `partfrac`, except that it always finds the partial fraction expansion over \mathbb{C} .

Example.

Find the partial fraction expansion of:

$$\frac{x^5 - 2x^3 + 1}{x^4 - 2x^3 + 2x^2 - 2x + 1}$$

over the real numbers. *Input (in real mode):*

```
partfrac((x^5-2*x^3+1)/(x^4-2*x^3+2*x^2-2*x+1))
```

Output:

$$x + 2 - \frac{1}{2(x-1)} + \frac{x-3}{2(x^2+1)}$$

To find the partial fraction decomposition over the complex numbers, you can either put Xcas in complex mode (see Section 3.5.5 p.72) or use `cpartfrac`.

Input (in complex mode):

```
partfrac((x^5-2*x^3+1)/(x^4-2*x^3+2*x^2-2*x+1))
```

or, in real or complex mode:

```
cpartfrac((x^5-2*x^3+1)/(x^4-2*x^3+2*x^2-2*x+1))
```

Output:

$$x + 2 - \frac{1}{2(x-1)} + \frac{-1-2i}{(2-2i)(x+i)} + \frac{2+i}{(2-2i)(x-i)}$$

6.33 Exact roots and poles

6.33.1 Roots and poles of a rational function: `froot`

The `froot` command finds roots and poles of a rational function.

- `froot` takes one argument:
rat, a rational function.
- `froot(rat)` returns a vector whose components are the roots and the poles of *rat*, each one followed by its multiplicity.
If Xcas can not find the exact values of the roots or poles, it tries to find approximate values if *rat* has numeric coefficients.

Examples.

- *Input:*

```
froot((x^5-2*x^4+x^3)/(x-2))
```

Output:

```
[1, 2, 0, 3, 2, -1]
```

Hence, for $F(x) = \frac{x^5 - 2x^4 + x^3}{x - 2}$:

- 1 is a root of multiplicity 2,
- 0 is a root of multiplicity 3,
- 2 is a pole of order 1.

• *Input:*

```
froot((x^3-2*x^2+1)/(x-2))
```

Output:

```
[1, 1, (sqrt(5)+1)/2, 1, (-sqrt(5)+1)/2, 1, 2, -1]
```

Remark.

To find the complex roots and poles, put **Xcas** in complex mode (check **Complex** in the **cas** configuration, red button giving the state line; see Section 3.5.5 p.72).

Example.

Input (in complex mode):

```
froot((x^2+1)/(x-2))
```

Output:

```
[-i, 1, i, 1, 2, -1]
```

6.33.2 Rational function given by roots and poles: fcoeff

The **fcoeff** command finds a rational function given its roots and poles.

- **fcoeff** takes one argument:
roots, a list consisting of the roots and poles of a rational function, each one followed by its multiplicity.
- **fcoeff(roots)** returns the rational function with the given roots and poles.

Example.

Input:

```
fcoeff([1, 2, 0, 3, 2, -1])
```

Output:

$$(x - 1)^2 x^3 (x - 2)^{-1}$$

6.34 Computing in $\mathbb{Z}/p\mathbb{Z}$ or in $\mathbb{Z}/p\mathbb{Z}[x]$

The way to compute over $\mathbb{Z}/p\mathbb{Z}$ or over $\mathbb{Z}/p\mathbb{Z}[x]$ depends on the syntax mode:

- In **Xcas** mode, an object n over $\mathbb{Z}/p\mathbb{Z}$ is written $n\%p$.
The representation is the symmetric representation:
 $11\%13$ returns $-2\%13$.

Examples.

- An integer n in $\mathbb{Z}/13\mathbb{Z}$
 $n:=12\%13$.
- a vector V in $\mathbb{Z}/13\mathbb{Z}$
 $V:=[1,2,3]\%13$ or $V:=[1\%13,2\%13,3\%13]$.
- a matrix A in $\mathbb{Z}/13\mathbb{Z}$
 $A:=[[1,2,3],[2,3,4]]\%13$ or
 $A:=[[1\%13,2\%13,3\%13],[2\%13,3\%13,4\%13]]$.
- a polynomial A in $\mathbb{Z}/13\mathbb{Z}[x]$ in symbolic representation
 $A:=(2*x^2+3*x-1)\%13$ or
 $A:=2\%13*x^2+3\%13*x-1\%13$.
- a polynomial A in $\mathbb{Z}/13\mathbb{Z}[x]$ in list representation
 $A:=poly1[1,2,3]\%13$ or $A:=poly1[1\%13,2\%13,3\%13]$.

To recover an object o with integer coefficients instead of modular coefficients, input $o \% 0$. For example:

Input: $o:=4\%7; : o\%0$ *Output:*

–3

Remark. Most **Xcas** functions that work on integers or polynomials with integer coefficients will often work the same on $\mathbb{Z}/p\mathbb{Z}$ or $\mathbb{Z}/p\mathbb{Z}[x]$, with the obvious exception that the input and output will be modular. They will be listed in the remaining subsections. For some commands in $\mathbb{Z}/p\mathbb{Z}$ or in $\mathbb{Z}/p\mathbb{Z}[x]$, p must be a prime integer.

- In **Maple** mode, integers modulo p are represented like usual integers instead of using specific modular integers. To avoid confusion with normal commands, modular commands are written with a capital letter (inert form) and followed by the mod command.

The Maple commands will be discussed in Section 6.35 p.433.

6.34.1 Expanding and reducing: normal

The **normal** command expands and reduces expressions in $\mathbb{Z}/p\mathbb{Z}[x]$. (See also Section 6.12.13 p.212.)

- **normal** takes one argument:
 $expr$, a modular expression.
- **normal($expr$)** returns the expanded irreducible representation of $expr$.

Example.*Input:*

```
normal(((2*x^2+12)*( 5*x-4))%13)
```

Output:

$$((-3) \% 13) x^3 + (5 \% 13) x^2 + ((-5) \% 13) x + 4 \% 13$$

6.34.2 Addition in $\mathbb{Z}/p\mathbb{Z}$ or in $\mathbb{Z}/p\mathbb{Z}[x]$: +

The + operator adds two integers in $\mathbb{Z}/p\mathbb{Z}$ or two polynomials in $\mathbb{Z}/p\mathbb{Z}[x]$. (See also Section 6.8.2 p.173.) For polynomial expressions, use the **normal** command to simplify.

Examples.

- For integers in $\mathbb{Z}/p\mathbb{Z}$:

Input:

```
3%13+10%13
```

Output:

```
0 % 13
```

- For polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$:

Input:

```
normal((11*x+5 )% 13+(8*x+6)%13)
```

or:

```
normal((11% 13*x+5%13)+(8% 13*x+6%13))
```

Output:

$$(6 \% 13) x + (-2) \% 13$$

6.34.3 Subtraction in $\mathbb{Z}/p\mathbb{Z}$ or in $\mathbb{Z}/p\mathbb{Z}[x]$: -

The - operator subtracts two integers in $\mathbb{Z}/p\mathbb{Z}$ or two polynomials in $\mathbb{Z}/p\mathbb{Z}[x]$. (See also Section 6.8.2 p.173.) For polynomial expressions, use the **normal** command to simplify.

Examples.

- For integers in $\mathbb{Z}/p\mathbb{Z}$:

Input:

```
31%13-10%13
```

Output:

$$(-5) \% 13$$

- For polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$:

Input:

```
normal((11*x+5)%13-(8*x+6)%13)
```

or:

```
normal(11% 13*x+5%13-(8% 13*x+6%13))
```

Output:

$$(3 \% 13)x + (-1) \% 13$$

6.34.4 Multiplication in $\mathbb{Z}/p\mathbb{Z}$ or in $\mathbb{Z}/p\mathbb{Z}[x]$: *

The `*` operator multiplies two integers in $\mathbb{Z}/p\mathbb{Z}$ or two polynomials in $\mathbb{Z}/p\mathbb{Z}[x]$. (See also Section 6.8.2 p.173.) For polynomial expressions, use the `normal` command to simplify.

Examples.

- For integers in $\mathbb{Z}/p\mathbb{Z}$:

Input:

```
31%13*10%13
```

Output:

$$(-2) \% 13$$

- For polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$:

Input:

```
normal((11*x+5)%13*(8*x+6 )% 13)
```

or:

```
normal((11% 13*x+5%13)*(8% 13*x+6%13))
```

Output:

$$((-3) \% 13)x^2 + ((-24) \% 13)x + 17 \% 13$$

6.34.5 Euclidean quotient : quo

The `quo` command finds the quotient of two polynomials (see also Section 6.28.2 p.378).

- `quo` takes two mandatory arguments and one optional argument:
 - P and Q , two polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$.
 - Optionally x , the variable (by default \mathbf{x}), if P and Q are given as expressions.
- `quo($P, Q \langle, x \rangle$)` returns the Euclidean quotient of P divided by Q .

Example.*Input:*

$$\text{quo}((x^3+x^2+1)\%13, (2*x^2+4)\%13)$$
Output:

$$((-6)\%13)x + (-6)\%13$$

Indeed $x^3 + x^2 + 1 = (2x^2 + 4)(\frac{x+1}{2}) + \frac{5x-4}{4}$ and $-3 * 4 = -6 * 2 = 1 \pmod{13}$.

6.34.6 Euclidean remainder: rem

The **rem** command finds the remainder of the Euclidean division of two polynomials (see also Section 6.28.3 p.380).

- **rem** takes two mandatory arguments and one optional argument:
 - P and Q , two polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$.
 - Optionally x , the variable (by default x), if P and Q are given as expressions.
- **rem**($P, Q \langle, x \rangle$) returns the remainder of the Euclidean division of P divided by Q .

Example.*Input:*

$$\text{rem}((x^3+x^2+1)\%13, (2*x^2+4)\%13)$$
Output:

$$((-2)\%13)x + (-1)\%13$$

Indeed $x^3 + x^2 + 1 = (2x^2 + 4)(\frac{x+1}{2}) + \frac{5x-4}{4}$ and $-3 * 4 = -6 * 2 = 1 \pmod{13}$.

6.34.7 Euclidean quotient and euclidean remainder: quorem

The **quorem** command finds the quotient and remainder of the Euclidean division of two polynomials (see also Section 6.5.10 p.143 and Section 6.28.4 p.381).

- **quorem** takes two mandatory arguments and one optional argument:
 - P and Q , two polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$.
 - Optionally x , the variable (by default x), if P and Q are given as expressions.
- **quorem**($P, Q \langle, x \rangle$) returns the list of the quotient and remainder of the Euclidean division of P and Q .

Example.*Input:*

```
quorem((x^3+x^2+1)%13,(2*x^2+4)%13)
```

Output:

$$[((-6) \% 13)x + (-6) \% 13, ((-2) \% 13)x + (-1) \% 13]$$

Indeed $x^3 + x^2 + 1 = (2x^2 + 4)\left(\frac{x+1}{2}\right) + \frac{5x-4}{4}$
and $-3 * 4 = -6 * 2 = 1 \pmod{13}$.

6.34.8 Division in $\mathbb{Z}/p\mathbb{Z}$ or in $\mathbb{Z}/p\mathbb{Z}[x]$: /

The / operator divides two integers in $\mathbb{Z}/p\mathbb{Z}$ or two polynomials A and B in $\mathbb{Z}/p\mathbb{Z}[x]$. (See also Section 6.8.2 p.173.) Since $\mathbb{Z}/p\mathbb{Z}$ is only a field if p is prime, the quotient is only guaranteed to exist if p is prime (unless the denominator is 0 (mod p)).

- For integers in $\mathbb{Z}/p\mathbb{Z}$:

Example.

– *Input:*

```
5%13/2% 13
```

Since 13 is prime, you get:

Output:

```
(-4) % 13
```

– *Input:*

```
5%14/3% 14
```

Since 3 (mod 14) is invertible in $\mathbb{Z}/14\mathbb{Z}$, you get:

Output:

```
(-3) % 14
```

– *Input:*

```
5%14/7% 14
```

Since 7 (mod 14) is not invertible in $\mathbb{Z}/14\mathbb{Z}$, you will get an error:

Output:

```
Not invertible Error: Bad Argument Value
```

- For polynomials, the result of P/Q is its irreducible representation in $\mathbb{Z}/p\mathbb{Z}[x]$.

Example.

Input:

```
(2*x^2+5)%13/(5*x^2+2*x-3)%13
```

Output:

$$\frac{(6 \% 13)x + 1 \% 13}{(2 \% 13)x + (2 \% 13) \% 13}$$

6.34.9 Power in $\mathbb{Z}/p\mathbb{Z}$ and in $\mathbb{Z}/p\mathbb{Z}[x]$: \wedge

The \wedge operator raises modular numbers and polynomials to powers in $\mathbb{Z}/p\mathbb{Z}$. (See also Section 6.8.2 p.173.) For polynomial expressions, use the **normal** command to simplify. **Xcas** uses the binary power algorithm to compute this.

Examples.

- *Input:*

$(5\%13)\wedge 2$

Output:

$(-1)\%13$

- *Input:*

`normal(((2*x+1)%13)^5)`

Output:

$(6\%13)x^5 + (2\%13)x^4 + (2\%13)x^3 + (1\%13)x^2 + ((-3)\%13)x + 1\%13$

because $10 = -3 \pmod{13}$, $40 = 1 \pmod{13}$, $80 = 2 \pmod{13}$, $32 = 6 \pmod{13}$.

6.34.10 Computing $a^n \bmod p$: `powmod` `powermod`

For integers a, n and p , the **powmod** finds $a^n \bmod p$.

powermod is a synonym for **powmod**.

- **powmod** takes three arguments:
 a, n and p , integers.
- **powmod**(a, n, p) returns $a^n \bmod p$ in $[0, p-1]$.

Examples.

- *Input:*

`powmod(5,2,13)`

Output:

12

- *Input:*

`powmod(5,2,12)`

Output:

1

6.34.11 Inverse in $\mathbb{Z}/p\mathbb{Z}$: `inv` `inverse` /

The `inv` command finds the inverse of an integer in $\mathbb{Z}/p\mathbb{Z}$.

`inverse` is a synonym for `inv`.

Since $\mathbb{Z}/p\mathbb{Z}$ is only a field if p is prime, the inverse is only guaranteed to exist if p is prime (and the integer is non-zero).

- `inv` takes one argument:
 $n\%p$, an element of $\mathbb{Z}/p\mathbb{Z}$.
- `inv($n\%p$)` returns the reciprocal of $n\%p$ in $\mathbb{Z}/p\mathbb{Z}$.

Example.

Input:

```
inv(3%13)
```

Output:

```
(-4) % 13
```

Indeed $3 \times -4 = -12 = 1 \pmod{13}$.

You can also find the reciprocal using division: *Input:*

```
1/(3%13)
```

Output:

```
(-4) % 13
```

6.34.12 Rebuilding a fraction from its value modulo p : `fracmod` `iratrecon`

Given an integer n and a modulus p , the `fracmod` (or `iratrecon`, for Maple compatibility) command finds the rational number equal to $n \bmod p$, where both the numerator and denominator are not greater than $\sqrt{p}/2$ in absolute value.

- `fracmod` (or `iratrecon`) takes two arguments:
 - n , an integer (representing a fraction).
 - p , an integer (the modulus).
- `fracmod(n, p)` (or `iratrecon(n, p)`) returns, if possible, a fraction a/b such that

$$\begin{aligned} a &= n \times b \pmod{p} \\ -\frac{\sqrt{p}}{2} &< a \leq \frac{\sqrt{p}}{2} \\ 0 &\leq b < \frac{\sqrt{p}}{2} \end{aligned}$$

In other words, $n = a/b \pmod{p}$.

Examples.

- *Input:*

```
fracmod(3,13)
```

Output:

$$-\frac{1}{4}$$

Indeed: $3 * -4 = -12 = 1 \pmod{13}$, hence $3 = -1/4 \% 13$.

Note that this means:

Input:

```
-1/4 % 13
```

Output:

```
3 % 13
```

- *Input:*

```
fracmod(13,121)
```

Output:

$$-\frac{4}{9}$$

Indeed: $13 \times -9 = -117 = 4 \pmod{121}$ hence $13 = -4/9 \% 13$.

6.34.13 GCD in $\mathbb{Z}/p\mathbb{Z}[x]$: gcd

The `gcd` command finds the greatest common divisor of two polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$ (for prime p). (See also Section 6.5.1 p.133 and Section 6.28.5 p.381.)

- `gcd` takes two arguments:
 P and Q , two polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$ (p must be prime).
- `gcd(P, Q)` returns the GCD of P and Q computed in $\mathbb{Z}/p\mathbb{Z}[x]$

Example.

Input:

```
gcd((2*x^2+5)%13, (5*x^2+2*x-3)%13)
```

Output:

```
(1 % 13) x + 2 % 13
```

6.34.14 Factoring over $\mathbb{Z}/p\mathbb{Z}[x]$: factor factoriser

The **factor** command factors polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$. (See also Section 6.12.10 p.209.)

- **factor** takes one argument:
 P , a polynomial with coefficients in $\mathbb{Z}/p\mathbb{Z}$ (p must be prime).
- **factor**(P) returns P in factored form.

Example.

Input:

```
factor((-3*x^3+5*x^2-5*x+4)%13)
```

Output:

```
((-3) % 13) ((1 % 13) x + (-6) % 13) ((1 % 13) x^2 + 6 % 13)
```

6.34.15 Determinant of a matrix in $\mathbb{Z}/p\mathbb{Z}$: det

The **det** command can find the determinant of a matrix with elements in $\mathbb{Z}/p\mathbb{Z}$. (See also Section 6.47.4 p.542.)

- **det** takes one argument:
 A , a matrix with elements in $\mathbb{Z}/p\mathbb{Z}$.
- **det**(A) returns the determinant of A .
 Computations are done in $\mathbb{Z}/p\mathbb{Z}$ by Gaussian reduction.

Example.

Input:

```
det([[1,2,9]%13,[3,10,0]%13,[3,11,1]%13])
```

or:

```
det([[1,2,9],[3,10,0],[3,11,1]]%13)
```

Output:

```
5 % 13
```

```
5%13
```

Hence, in $\mathbb{Z}/13\mathbb{Z}$, the determinant of $M = [[1, 2, 9], [3, 10, 0], [3, 11, 1]]$ is 5%13 (in \mathbb{Z} , $\det(M)=31$).

6.34.16 Inverse of a matrix with coefficients in $\mathbb{Z}/p\mathbb{Z}$: `inv` `inverse`

The `inv` command can find the inverse of a matrix with elements in $\mathbb{Z}/p\mathbb{Z}$. (See also Section 6.47.2 p.542.)
`inverse` is a synonym for `inv`.

- `inv` takes one argument:
 A , a matrix in $\mathbb{Z}/p\mathbb{Z}$.
- `inv(A)` returns the inverse of the matrix A .

Example.

Input:

```
inv([[1,2,9]%13,[3,10,0]%13,[3,11,1]%13])
```

or:

```
inverse([[1,2,9]%13,[3,10,0]%13,[3,11,1]%13])
```

or:

```
inv([[1,2,9],[3,10,0],[3,11,1]]%13)
```

or:

```
inverse([[1,2,9],[3,10,0],[3,11,1]]%13)
```

Output:

$$\begin{bmatrix} 2 \% 13 & (-4) \% 13 & (-5) \% 13 \\ 2 \% 13 & 0 \% 13 & (-5) \% 13 \\ (-2) \% 13 & (-1) \% 13 & 6 \% 13 \end{bmatrix}$$

6.34.17 Row reduction to echelon form in $\mathbb{Z}/p\mathbb{Z}$: `rref`

The `rref` command can find the reduced row echelon form of a matrix with elements in $\mathbb{Z}/p\mathbb{Z}$. (See 6.56.3):

- `rref` takes one argument:
 A , a matrix in $\mathbb{Z}/p\mathbb{Z}$.
- `rref(A)` returns the echelon form of A .

Example.

Input:

```
rref([[0, 2, 9]%15,[1,10,1]%15,[2,3,4]%15])
```

Output:

$$\begin{bmatrix} 1 \% 15 & 0 \% 15 & 0 \% 15 \\ 0 \% 15 & 1 \% 15 & 0 \% 15 \\ 0 \% 15 & 0 \% 15 & 1 \% 15 \end{bmatrix}$$

This can be used to solve a linear system of equations with coefficients in $\mathbb{Z}/p\mathbb{Z}$ by rewriting it in matrix form

$$A \cdot X = B$$

`rref` can then take as argument the augmented matrix of the system (the matrix obtained by augmenting matrix A to the right with the column vector B).

`rref` will return a matrix $[A_1, B_1]$ where A_1 has 1s on its principal diagonal and zeros outside. The solutions in $\mathbb{Z}/p\mathbb{Z}$ of:

$$A_1 \cdot X = B_1$$

are the same as the solutions of:

$$A \cdot X = B$$

$$A \cdot X = B$$

Example.

Solve in $\mathbb{Z}/13\mathbb{Z}$

$$\begin{cases} x + 2 \cdot y = 9 \\ 3 \cdot x + 10 \cdot y = 0 \end{cases}$$

Input:

```
rref([[1, 2, 9]%13,[3,10,0]%13])
```

or:

```
rref([[1, 2, 9],[3,10,0]])%13
```

Output:

$$\begin{bmatrix} 1 \% 13 & 0 \% 13 & 3 \% 13 \\ 0 \% 13 & 1 \% 13 & 3 \% 13 \end{bmatrix}$$

hence $x = 3 \% 13$ and $y = 3 \% 13$.

6.34.18 Construction of a Galois field: GF

A *Galois field* is a finite field. A Galois field will have characteristic p for some prime number p , and the order will be p^n for some integer n . Any Galois field of order p^n will be isomorphic to $\mathbb{Z}/p\mathbb{Z}[X]/I$, where I is the ideal generated by an irreducible polynomial $P(X)$ in $\mathbb{Z}/p\mathbb{Z}[X]$

The `GF` command creates Galois fields.

- `GF` takes two mandatory arguments and one optional argument:
 - p , a prime number.
 - n , an integer greater than 1 (or an irreducible polynomial over $\mathbb{Z}/p\mathbb{Z}[X]$).
 If n is an integer, the first two arguments can be combined and entered as a prime power p^n .

- Optionally *vars*, either the name of a variable or a list of two or three variables. These variables must be symbolic, so you should purge them if necessary.
- $\text{GF}(p, n \langle vars \rangle)$ returns a Galois field of characteristic p having p^n elements. The output will look like $\text{GF}(p, P(k), [k, K, g], \text{undef})$ where:
 - p is the characteristic.
 - $P(k)$ is an irreducible polynomial generating an ideal I in $\mathbb{Z}/p\mathbb{Z}[X]$, the Galois field being the quotient of $\mathbb{Z}/p\mathbb{Z}[X]$ by I .
 - k is the name of the polynomial variable.
 - K is the name of the Galois field (which will be given to a free variable).
 - g is a generator of the multiplicative group K^* . You can build elements of the field with polynomials in g .

If the optional argument *vars* is given:

- *vars* consists of a variable name, then g is that variable name.
- If *vars* consists of a pair of variable names, then k will be the first variable and K will be the second variable.
In this case, there is no generator given and the elements of K must be given by $K(P(k))$ for a polynomial $P(k)$.
- If *vars* consists of three variable names, then k will be the first variable, K will be the second variable and g will be the third variable.

The elements of the field will be $0, g, g^2, \dots, g^{p^n-2}$.

Example.

Input:

$\text{GF}(2, 8)$

Output:

$\text{GF}(2, k^8 + k^4 + k^3 + k^2 + 1, [k, K, g], \text{undef})$

The field K has $2^8 = 256$ elements and g generates the multiplicative group of this field ($\{1, g, g^2, \dots, g^{254}\}$).

The elements of this field can be written as polynomials in g or as $K(P(k))$, where $P(k)$ is a polynomial in k . *Input:*

g^9

or: Input:

$K(k^9)$

or: Output:

$(g^5 + g^4 + g^3 + g)$

indeed $g^8 = g^4 + g^3 + g^2 + 1$, so $g^9 = g^5 + g^4 + g^3 + g$.

Once a Galois field is created in **Xcas**, you can use elements of the field to create polynomials and matrices, and use the usual operators on them, such as `+`, `-`, `*`, `/`, `^`, `inv`, `sqrt`, `quo`, `rem`, `quorem`, `diff`, `factor`, `gcd`, `egcd`, etc.

Examples.

- Compute the inverse of a matrix in a Galois field:

Input:

```
GF(3,5,b):: A:= [[1,b],[b,1]]:: inv(A)
```

Output:

$$\begin{bmatrix} (b^3 + b^2 - b) & (-b^4 - b^3 + b^2) \\ (-b^4 - b^3 + b^2) & (b^3 + b^2 - b) \end{bmatrix}$$

- Factor a polynomial over a Galois field:

Input:

```
GF(5,3,c):: p:= x^2-c-1:: factor(p)
```

Output:

$$((1 \% 5)x + ((-2 \cdot c^2 + 2 \cdot c))) ((1 \% 5)x + ((2 \cdot c^2 - 2 \cdot c)))$$

There are still some limitations due to an incomplete implementation of some algorithms, such as multivariate factorization when the polynomial is not unitary.

Example:

Input:

```
G(x)^255
```

Output should be the unit, indeed:

```
GF(2,x^8-x^6-x^4-x^3-x^2-x-1,x,1)
```

As one can see in these examples, the output contains many times the same information that you would prefer not to see if you work many times with the same field. For this reason, the definition of a Galois field may have an optional argument, a variable name which will be used thereafter to represent elements of the field. Since you will also most likely want to modify the name of the indeterminate, the field name is grouped with the variable name in a list passed as third argument to **GF**. Note that these two variable names must be quoted.

Example.

Input:

```
G:=GF(2,2,['w'],'G'):: G(w^2)
```

Output:

```
Done, G(w+1)
```

Input:

$$G(w^3)$$

Output:

$$G(1)$$

Hence, the elements of $GF(2, 2)$ are $G(0), G(1), G(w), G(w^2)=G(w+1)$.

We may also impose the irreducible primitive polynomial that we wish to use, by putting it as second argument (instead of n), for example:

$$G := GF(2, w^8 + w^6 + w^3 + w^2 + 1, ['w', 'G'])$$

If the polynomial is not primitive, **Xcas** will replace it automatically by a primitive polynomial, for example:

Input:

$$G := GF(2, w^8 + w^7 + w^5 + w + 1, ['w', 'G'])$$

Output:

$$G := GF(2, w^8 - w^6 - w^3 - w^2 - 1, ['w', 'G'], \text{undef})$$

6.34.19 Factoring a polynomial with coefficients in a Galois field: **factor**

The **factor** command can factor univariate polynomials with coefficients in a Galois field.

- **factor** takes one mandatory argument and one optional argument:
 - *expr*, an expression or a list of expressions.
 - Optionally, α , to specify an extension field.
- **factor(expr)** returns *expr* factored over the field of its coefficients, with the addition of i in complex mode (see Section 3.5.5 p.72). If **sqrt** is enabled in the Cas configuration (see Section 3.5.7 p.73), polynomials of order 2 are factorized in complex mode or in real mode if the discriminant is positive.
factor(expr, α) returns *expr* factored over $F[\alpha]$, where F is the field of coefficients of *expr*.
- **cfactor** factors like **factor**, except the field includes i whether in real or complex mode.

Examples.

factor can also factorize a univariate polynomial with coefficients in a Galois field.

Input (for example to have $G = \mathbb{F}_4$):

$$G := GF(2, 2, ['w', 'G'])$$

Output:


```
GF(2,w^2+w+1,[w,G],undef)
```

Input (for example):

```
a:=G(w)
factor(a^2*x^2+1)
```

Output:

```
(G(w+1))*(x+G(w+1))^2
```

6.35 Computing in $\mathbb{Z}/p\mathbb{Z}[x]$ using Maple syntax

You can set `Xcas` to work in `Maple` mode rather than native `Xcas` mode (see Section 3.5.2 p.71).

6.35.1 Euclidean quotient: Quo

In `Xcas` mode, `Quo` is simply the inert form of `quo`; namely, it returns the Euclidean quotient of two polynomials without evaluation. (See section Section 6.28.2 p.378.) In `Maple` mode, the `Quo` command can additionally be used in conjunction with `mod` to compute the Euclidean quotient of two polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$.

- (In `Maple` mode.)
`Quo` takes two arguments:
 P and Q , two polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$.
- `Quo(P, Q)` returns the Euclidean quotient of P divided by Q .

Examples.

- *Input (in Xcas mode):*

```
Quo((x^3+x^2+1) mod 13,(2*x^2+4) mod 13)
```

Output:

```
quo((1 % 13) x^3 + (1 % 13) x^2 + 1 % 13, (2 % 13) x^2 + 4 % 13)
```

To get the result of the division:

Input:

```
eval(ans())
```

```
((-6) % 13) x + (-6) % 13
```

Input (in Maple mode):

```
Quo(x^3+x^2+1,2*x^2+4) mod 13
```

Output:

```
-6x - 6
```

- *Input (in Maple mode):*

$$\text{Quo}(x^2+2x, x^2+6x+5) \bmod 5$$

Output:

$$1$$

6.35.2 Euclidean remainder: Rem

In Xcas mode, **Rem** is simply the inert form of **rem**; namely, it returns the Euclidean remainder of two polynomials without evaluation. (See section Section 6.28.3 p.380.) In Maple mode, the **rem** command can additionally be used in conjunction with **mod** to compute the Euclidean remainder of two polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$.

- (In Maple mode.)
Rem takes two arguments:
 P and Q , two polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$.
- **Rem**(P, Q) returns the Euclidean remainder of P divided by Q .

Examples:

- *Input (in Xcas mode):*

$$\text{Rem}(x^3+x^2+1 \bmod 13, (2x^2+4) \bmod 13)$$

Output:

$$\text{rem}((1 \% 13)x^3 + (1 \% 13)x^2 + 1 \% 13, (2 \% 13)x^2 + 4 \% 13)$$

To get the result of the division:

Input:

$$\text{eval}(\text{ans}())$$

Output:

$$((-2) \% 13)x + (-1) \% 13$$

- *Input (in Maple mode):*

$$\text{Rem}(x^3+x^2+1, 2x^2+4) \bmod 13$$

Output:

$$-2x - 1$$

- *Input (in Maple mode):*

$$\text{Rem}(x^2+2x, x^2+6x+5) \bmod 5$$

Output:

$$x$$

6.35.3 GCD in $\mathbb{Z}/p\mathbb{Z}[x]$: Gcd

In **Xcas** mode, **Gcd** is simply the inert form of **gcd**; namely, it returns the greatest common divisor of two polynomials without evaluation. (See section Section 6.28.5 p.381.) In **Maple** mode, the **Gcd** command can additionally be used in conjunction with **mod** to compute the greatest common divisor of two polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$.

- (In **Maple** mode.)
Gcd takes an unspecified number of arguments:
polys, a sequence or list of polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$.
- **Gcd(polys)** returns the greatest common divisor of the polynomials in *polys*.

Examples.

- *Input (in Xcas mode):*

$$\text{Gcd}(2*x^2+5\%13, 5*x^2+2*x-3\%13)$$

Output:

$$\text{gcd}(2x^2 + 5 \% 13, 5x^2 + 2x + (-3) \% 13)$$

To get the actual greatest common divisor:

Input:

$$\text{eval}(\text{ans}())$$

Output:

$$(1 \% 13)x + 2 \% 13$$

Input (in Maple mode):

$$\text{Gcd}(2*x^2+5, 5*x^2+2*x-3) \text{ mod } 13$$

Output:

$$x + 2$$

- *Input (in Maple mode):*

$$\text{Gcd}(x^2+2*x, x^2+6*x+5) \text{ mod } 5$$

Output:

$$x$$

6.35.4 Factoring in $\mathbb{Z}/p\mathbb{Z}[x]$: **Factor**

In **Xcas** mode, **Factor** is simply the inert form of **factor**; namely, it factors a polynomial without evaluation. (See section Section 6.12.10 p.209.) In **Maple** mode, the **Factor** command can additionally be used in conjunction with **mod** to factor a polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$, where p must be prime.

- (In **Maple** mode.)
Factor takes one argument:
 P , a polynomial with coefficients in $\mathbb{Z}/p\mathbb{Z}$ for prime p .
- **Factor**(P) returns the factored form of P .

Example.

Input (in Xcas mode):

```
Factor((-3*x^3+5*x^2-5*x+4)%13)
```

Output:

```
factor (((-3) % 13) x^3 + (5 % 13) x^2 + ((-5) % 13) x + 4 % 13)
```

To get the actual factorization:

Input:

```
eval(ans())
```

Output:

```
((-3) % 13) ((1 % 13) x + (-6) % 13) ((1 % 13) x^2 + 6 % 13)
```

Input (in Maple mode):

```
Factor(-3*x^3+5*x^2-5*x+4) mod 13
```

Output:

```
-3(x - 6)(x^2 + 6)
```

6.35.5 Determinant of a matrix with coefficients in $\mathbb{Z}/p\mathbb{Z}$: **Det**

In **Xcas** mode, **Det** is simply the inert form of **det**; namely, it gives the determinant of a matrix without evaluating it. (See section Section 6.47.4 p.542.) In **Maple** mode, the **Det** command can additionally be used in conjunction with **mod** to find the determinant of a matrix whose elements are in $\mathbb{Z}/p\mathbb{Z}$.

- (In **Maple** mode.)
Det takes one argument:
 A , a matrix with elements in $\mathbb{Z}/p\mathbb{Z}$.
- **Det**(A) returns the determinant of A .

Example.

Input (in Xcas mode):

```
Det([[1,2,9] mod 13,[3,10,0] mod 13,[3,11,1] mod 13])
```

Output:

$$\det \begin{pmatrix} 1 \% 13 & 2 \% 13 & (-4) \% 13 \\ 3 \% 13 & (-3) \% 13 & 0 \% 13 \\ 3 \% 13 & (-2) \% 13 & 1 \% 13 \end{pmatrix}$$

To find the value of the determinant, you can enter:

Input:

```
eval(ans())
```

Output:

5 % 13

Hence, in $\mathbb{Z}/13\mathbb{Z}$, the determinant of $A = [[1, 2, 9], [3, 10, 0], [3, 11, 1]]$ is 5%13 (in \mathbb{Z} , $\det(A)=31$).

Input (in Maple mode):

```
Det([[1,2,9],[3,10,0],[3,11,1]]) mod 13
```

Output:

5

6.35.6 Inverse of a matrix in $\mathbb{Z}/p\mathbb{Z}$: Inverse

In Xcas mode, **Inverse** is simply the inert form of **inverse**; namely, it gives the inverse of a matrix without evaluating it. (See section Section 6.47.2 p.542.) In Maple mode, the **Inverse** command can additionally be used in conjunction with **mod** to find the inverse of a matrix whose elements are in $\mathbb{Z}/p\mathbb{Z}$.

- (In Maple mode.)
Inverse takes one argument:
 A , a matrix with elements in $\mathbb{Z}/p\mathbb{Z}$.
- **Det**(A) returns the inverse of A .

Example.

Input (in Xcas mode):

```
Inverse([[1,2,9] mod 13,[3,10,0] mod 13,[3,11,1] mod13])
```

Output:

$$\text{inverse} \left(\begin{pmatrix} 1 \% 13 & 2 \% 13 & (-4) \% 13 \\ 3 \% 13 & (-3) \% 13 & 0 \% 13 \\ 3 \% 13 & (-2) \% 13 & 1 \% 13 \end{pmatrix} \right)$$

To get the actual inverse, you can enter:

Input:

```
eval(ans())
```

Output:

$$\begin{bmatrix} 2 \% 13 & (-4) \% 13 & (-5) \% 13 \\ 2 \% 13 & 0 \% 13 & (-5) \% 13 \\ (-2) \% 13 & (-1) \% 13 & 6 \% 13 \end{bmatrix}$$

which is the inverse of $A = [[1, 2, 9], [3, 10, 0], [3, 11, 1]]$ in $\mathbb{Z}/13\mathbb{Z}$.

Input (in Maple mode):

```
Inverse([[1,2,9],[3,10,0],[3,11,1]]) mod 13
```

Output:

$$\begin{bmatrix} 2 & -4 & -5 \\ 2 & 0 & -5 \\ -2 & -1 & 6 \end{bmatrix}$$

6.35.7 Row reduction to echelon form in $\mathbb{Z}/p\mathbb{Z}$: Rref

In **Xcas** mode, **Rref** is simply the inert form of **rref**; namely, it returns **rref** without evaluating it. (See section Section 6.56.3 p.623.) In **Maple** mode, the **Rref** command can additionally be used in conjunction with **mod** to find the reduced row echelon form of a matrix whose elements are in $\mathbb{Z}/p\mathbb{Z}$.

- (In **Maple** mode.)
Rref takes one argument:
 A , a matrix with elements in $\mathbb{Z}/p\mathbb{Z}$.
- **Rref**(A) returns the reduced row echelon form of A .

Example.

Solve in $\mathbb{Z}/13\mathbb{Z}$:

$$\begin{cases} x + 2 \cdot y = 9 \\ 3 \cdot x + 10 \cdot y = 0 \end{cases}$$

Input (in Xcas mode):

```
Rref([[1,2,9] mod 13,[3,10,0] mod 13])
```

Output:

$$\text{rref} \left(\begin{bmatrix} 1 \% 13 & 2 \% 13 & (-4) \% 13 \\ 3 \% 13 & (-3) \% 13 & 0 \% 13 \end{bmatrix} \right)$$

To actually get the reduced echelon form, you can enter:

Input:

```
eval(ans())
```

Output:

$$\begin{bmatrix} 1 \% 13 & 0 \% 13 & 3 \% 13 \\ 0 \% 13 & 1 \% 13 & 3 \% 13 \end{bmatrix}$$

and conclude that $x=3\%13$ and $y=3\%13$.

Input (in Maple mode):

```
Rref([[1,2,9],[3,10,0]] mod 13)
```

Output:

$$\begin{bmatrix} 1 & 0 & 3 \\ 0 & 1 & 3 \end{bmatrix}$$

and again conclude that $x=3\%13$ and $y=3\%13$.

6.36 Taylor and asymptotic expansions

6.36.1 Dividing by increasing power order: `divpc`

The `divpc` command finds the truncated Taylor expansion of a quotient of polynomials.

- `divpc` takes three mandatory arguments and one optional argument:
 - P and Q , two polynomial expressions such that Q has a nonzero constant term/
 - n , an integer.
 - Optionally, x , the variable name (by default x).
- `divpc($P, Q, n \langle, x \rangle$)` returns the Taylor expansion of P/Q of order n about $x = 0$.

Note that this command does not work on polynomials written as a list of coefficients.

Example.

Input:

```
divpc(1+x^2+x^3,1+x^2,5)
```

Output:

$$-x^5 + x^3 + 1$$

6.36.2 Series expansion: `taylor series`

The `taylor` command finds Taylor expansions.

`series` is a synonym for `taylor`.

- `taylor` takes one mandatory and four optional arguments:
 - $expr$, an expression depending on a variable.
 - Optionally, x , the variable (by default x).
 - Optionally n , an integer, the order of the series expansion (by default 5).
 - Optionally, a , the center of the Taylor expansion (by default 0). This can be combined with the optional x by replacing x by $x = a$.
 - dir , a direction, which can be -1 or 1 , for unidirectional series expansion, or 0 (for bidirectional series expansion) (by default 0).
- `taylor($expr, x \langle, a, n, dir \rangle$)` returns the Taylor expansion of $expr$ about a or order n ; consisting of a polynomial in $x - a$ plus a remainder of the form of the form:

$$(x - a)^n * \text{order_size}(x - a)$$

where `order_size` is a function such that,

$$\forall r > 0, \quad \lim_{x \rightarrow 0} x^r \text{order_size}(x) = 0$$

For regular series expansion, `order_size` is a bounded function, but for non regular series expansion, it might tend slowly to infinity, for example like a power of $\ln(x)$.

Example.

Input:

```
taylor(sin(x),x=1,2)
```

or:

```
series(sin(x),x=1,2)
```

or (be careful with the order of the arguments!):

```
taylor(sin(x),x,2,1)
```

or:

```
series(sin(x),x,2,1)
```

Output:

$$\sin(1) + \cos(1)(x-1) - \frac{1}{2}\sin(1)(x-1)^2 + (x-1)^3 \text{order_size}(x-1)$$

Remark.

The order returned by `taylor` may be smaller than n if cancellations between numerator and denominator occur, for example consider

$$\frac{x^3 + \sin(x)^3}{x - \sin(x)}$$

Input:

```
taylor(x^3+sin(x)^3/(x-sin(x)),x=0,5)
```

Output:

$$6 - \frac{27}{10}x^2 + x^3 + \frac{711}{1400}x^4 + x^6 \text{order_size}(x)$$

$$6+ -27/10*x^2+x^3*\text{order_size}(x)$$

which is only a 2nd degree expansion. Indeed the numerator and denominator valuation is 3, hence you lose 3 orders. To get order 4, you should use $n = 7$.

Input:

```
taylor(x^3+sin(x)^3/(x-sin(x)),x=0,7)
```

Output:

$$6 - \frac{27}{10}x^2 + x^3 + \frac{711}{1400}x^4 - \frac{737}{14000}x^6 + x^8 \text{order_size}(x)$$

$$6+ -27/10*x^2+x^3+711/1400*x^4+x^5*\text{order_size}(x)$$

a fourth degree expansion.

Examples.

- Find a 4th-order expansion of $\cos(2x)^2$ in the vicinity of $x = \frac{\pi}{6}$.

Input:

```
taylor(cos(2*x)^2,x=pi/6, 4)
```

Output:

$$\frac{1}{4} - \sqrt{3} \left(x - \frac{\pi}{6}\right) + 2 \left(x - \frac{\pi}{6}\right)^2 + \frac{8}{3} \sqrt{3} \left(x - \frac{\pi}{6}\right)^3 - \frac{8}{3} \left(x - \frac{\pi}{6}\right)^4 + \left(x - \frac{\pi}{6}\right)^5 \text{order_size} \left(x - \frac{\pi}{6}\right)$$

- Find a 5th-order series expansion of $\arctan(x)$ in the vicinity of $x = +\infty$.

Input:

```
series(atan(x),x=+infinity,5)
```

Output:

$$\frac{\pi}{2} - \frac{1}{x} + \frac{\left(\frac{1}{x}\right)^3}{3} - \frac{\left(\frac{1}{x}\right)^5}{5} + \left(\frac{1}{x}\right)^6 \text{order_size} \left(\frac{1}{x}\right)$$

Note that the expansion variable and the argument of the `order_size` function is $h = \frac{1}{x} \rightarrow_{x \rightarrow +\infty} 0$.

- Find a 2nd-order expansion of $(2x - 1)e^{\frac{1}{x-1}}$ in the vicinity of $x = +\infty$.

Input:

```
series((2*x-1)*exp(1/(x-1)),x=+infinity,3)
```

Output (only a 1st-order series expansion):

$$2 \left(\frac{1}{x}\right)^{-1} + 1 + \frac{2}{x} + \frac{17}{6} \left(\frac{1}{x}\right)^2 + \left(\frac{1}{x}\right)^3 \text{order_size} \left(\frac{1}{x}\right)$$

Note that this is only a 1st order expansion. To get a 2nd-order series expansion in $1/x$:

Input:

```
series((2*x-1)*exp(1/(x-1)),x=+infinity,4)
```

Output:

$$2 \left(\frac{1}{x}\right)^{-1} + 1 + \frac{2}{x} + \frac{17}{6} \left(\frac{1}{x}\right)^2 + \frac{47}{12} \left(\frac{1}{x}\right)^3 + \left(\frac{1}{x}\right)^4 \text{order_size} \left(\frac{1}{x}\right)$$

- Find a 2nd-order series expansion of $(2x - 1)e^{\frac{1}{x-1}}$ in the vicinity of $x = -\infty$.

Input:

```
series((2*x-1)*exp(1/(x-1)),x=-infinity,4)
```

Output:

$$-2 \left(-\frac{1}{x}\right)^{-1} + 1 + \frac{2}{x} + \frac{17}{6} \left(-\frac{1}{x}\right)^2 - \frac{47}{12} \left(-\frac{1}{x}\right)^3 + \left(-\frac{1}{x}\right)^4 \text{ order_size} \left(-\frac{1}{x}\right)$$

- Find a 2nd-order series expansion of $\frac{(1+x)^{\frac{1}{x}}}{x^3}$ in the vicinity of $x = 0^+$.
Input:

```
series((1+x)^(1/x)/x^3,x=0,2,1)
```

(Note that this is a one-sided series expansion, since *dir*=1.) *Output:*

$$ex^{-3} - \frac{1}{2}ex^{-2} + x^{-1}\text{order_size}(x)$$

6.36.3 The inverse of a series: revert

The **revert** command finds the beginning of the power series of a function given the beginning of the series of the function.

- *revert* takes one mandatory and one optional argument:
 - *series*, the beginning of a power series centered at 0 for a function *f*.
 - Optionally *x*, the name of the variable (by default *x*).
- **revert**(*series* $\langle, x \rangle$) returns the beginning of the power series for the inverse of *f*; namely the beginning of the power series for $g(f(0) + x)$ where the function *g* satisfies $g(f(x)) = x$.

Examples.

- *Input:*

```
revert (x + x^2 + x^4)
```

Output:

$$x - x^2 + 2x^3 - 6x^4$$

Note that if the power series of a function *f* begins with $x + x^2 + x^4$, then $f(0) = 0$, $f'(0) = 1$, $f''(0) = 2$, $f'''(0) = 0$ and $f^{(4)}(0) = 24$. The function *g* with $g(f(x)) = x$ will then satisfy $g(0) = 0$, $g'(0) = 1/f'(0) = 1$, $g''(0) = -2$, $g'''(0) = 12$ and $g^{(4)}(0) = -144$. The power series for *g* will then begin $x - x^2 + 2x^3 - 6x^4$.

- *Input:*

```
revert(1 + x + x^2/2 + x^3/6 + x^4/24)
```

Note that the argument is the beginning of the power series for $\exp(x)$, so the output is the beginning of the power series for $\ln(1+x)$. *Output:*

$$x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4}$$

6.36.4 The residue of an expression at a point: residue

The `residue` command finds the residue of an expression at a point.

- `residue` takes three arguments:
 - *expr*, an expression depending on a variable.
 - *x*, a variable name.
 - *a*, a complex number. This can be combined with the previous argument in an equality $x = a$.
- `residue(expr, x, a)` returns the residue of *expr* at the point *a*.

Example.

Input:

```
residue(cos(x)/x^3,x,0)
```

or:

```
residue(cos(x)/x^3,x=0)
```

Output:

$$-\frac{1}{2}$$

6.36.5 Padé expansion: pade

The `pade` command finds a rational expression which agrees with a function up to a given order.

- `pade` takes 4 arguments
 - *expr*, an expression.
 - *x*, the variable name.
 - *n*, an integer or *R*, a polynomial.
 - *p*, an integer.
- `pade(expr, x, n, p)` or `pade(expr, x, P, p)` returns a rational function P/Q such that $\deg(P) < p$ and $P/Q = \text{expr} \pmod{x^{n+1}}$ (meaning that P/Q and *f* have the same Taylor expansion at 0 up to order *n*) or $P/Q = \text{expr} \pmod{R}$, respectively.

Examples.

- *Input:*

```
pade(exp(x), x, 5, 3)
```

or:

```
pade(exp(x), x, x^6, 3)
```

Output:

$$\frac{-3x^2 - 24x - 60}{x^3 - 9x^2 + 36x - 60}$$

To verify:

Input:

```
taylor((3*x^2+24*x+60)/(-x^3+9*x^2-36*x+60))
```

Output:

$$1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120} + x^6 \text{order_size}(x)$$

which is the 5th-order series expansion of `exp(x)` at $x = 0$.

• *Input:*

```
pade((x^15+x+1)/(x^12+1),x,12,3)
```

or:

```
pade((x^15+x+1)/(x^12+1),x,x^13,3)
```

Output:

$$x + 1$$

• *Input:*

```
pade((x^15+x+1)/(x^12+1),x,14,4)
```

or:

```
pade((x^15+x+1)/(x^12+1),x,x^15,4)
```

Output:

$$\frac{2x^3 + 1}{x^{11} - x^{10} + x^9 - x^8 + x^7 - x^6 + x^5 - x^4 + x^3 + x^2 - x + 1}$$

To verify:

Input:

```
series(ans(),x=0,15)
```

Output:

$$1 + x - x^{12} - x^{13} + 2x^{15} + x^{16} \text{order_size}(x)$$

Then:

Input:

```
series((x^15+x+1)/(x^12+1),x=0,15)
```

Output:

$$1 + x - x^{12} - x^{13} + x^{15} + x^{16} \text{order_size}(x)$$

These two expressions have the same 14th-order series expansion at $x = 0$.

6.37 Ranges of values

6.37.1 Definition of a range of values: ..

The `..` is an infix operator which sets a range of values; given two real numbers a and b , the range of values between them is denoted $a..b$.

Warning!

The order of the boundaries of the range is significant. For example, if you input

```
B:=2..3; C:=3..2,
```

then B and C are different; `B==C` returns 0.

Examples.

- *Input:*

```
1..4
```

Output:

```
1...4
```

- *Input:*

```
1.2..sqrt(2)
```

Output:

```
1.2...√2
```

Since `..` is an operator, the parts of an expression can be picked out of it (see Section 6.15.3 p.236). In particular, the `left` and `right` commands can find the left and right endpoints of a range (see Section 6.3.4 p.123, Section 6.15.3 p.236, Section 6.38.2 p.448, Section 6.40.6 p.467, Section 6.55.4 p.618 and Section 6.55.5 p.618 for other uses of `left` and `right`.)

Example:

Input:

```
R := 2..5
```

Then:

Input:

```
sommet(R)
```

Output:

```
..
```

Input:

```
left(R)
```

Output:

```
2
```

Input:

```
right(R)
```

Output:

```
5
```

6.37.2 Center of a range of values: `interval2center`

The `interval2center` command finds the midpoint of a range of values.

- `interval2center` takes one argument:
 R , a range of values interval or a list of ranges of values.
- `interval2center(R)` returns the center of this range or the list of centers of these ranges.

Examples.

- *Input:*

```
interval2center(3..5)
```

Output:

4

- *Input:*

```
interval2center([2..4,4..6,6..10])
```

Output:

[3, 5, 8]

6.37.3 Ranges of values defined by their center: `center2interval`

The `center2interval` command finds ranges of values determined by their centers.

- `center2interval` takes one mandatory argument and one optional argument:
 - L , a list of real numbers.
 - Optionally, c , a real number (by default $L[0] - (L[1] - L[0])/2$).
- `center2interval($L \langle, c \rangle$)` returns a list of ranges of values; the midpoints of the elements of the lists are endpoints of the ranges; so, for example, the second range will go from $(L[0] + L[1])/2$ to $(L[1] + L[2])/2$, etc. By default the first and last range are centered on $L[0]$ and $L[-1]$. With an argument of c , however, the first range will begin at c

Examples.

- *Input:*

```
center2interval([3,5,8])
```

Output:

2.0...4.0, 4.0...6.5, 6.5...9.5

- *Input:*

```
center2interval([3,5,8],2.5)
```

Output:

```
2.5...4.0,4.0...6.5,6.5...9.5
```

6.38 Intervals

6.38.1 Defining intervals: `i[]`

An interval is a range of real numbers, whose end points will be floats with at least 15 significant digits. The `i` command creates intervals, with the arguments in square brackets.

- `i` takes two arguments:
`a` and `b`, two real numbers.
- `i[a,b]` returns the interval between `a` and `b`.
If $a > b$, then `i[a,b]` returns `i[evalf(b,15)-epsilon,evalf(a,15)+epsilon]`
(see Section 3.5.7 p.73, item 9).

Examples.

- *Input:*

```
i[1,13/11]
```

Output:

```
[1.000000000000000..1.18181818181819]
```

- *Input:*

```
i[pi,sqrt(3)]
```

Output:

```
[1.73205080756886..3.14159265358980]
```

Intervals can also be created by following a decimal number with a question mark. If the decimal number contains n digits, the interval will be centered at a and have width $2 \cdot 10^{-n}$.

Examples.

- *Input:*

```
0.123?
```

Output:

```
[0.121999999999999..0.124000000000000]
```

Input:

```
789.123456?
```

Output:

```
[0.789123454999990e3..0.789123456999998e3]
```

6.38.2 The endpoints of an interval: `left` `right`

The `left` and `right` commands can find the left and right endpoints of an interval. (See Section 6.15.3 p.236, Section 6.40.6 p.467, Section 6.37.1 p.445, Section 6.55.4 p.618 and Section 6.55.5 p.618 for other uses of `left` and `right`.)

- `left` and `right` take one argument:
 I , an interval.
- `left(I)` returns the left endpoint of the interval I .
- `right(I)` returns the right endpoint of the interval I .

Examples.

- *Input:*

```
left(i[2,5])
```

Output:

```
2.0000000000000000
```

- *Input:*

```
right(i[2,5])
```

Output:

```
5.0000000000000000
```

6.38.3 Interval arithmetic: `+` `-` `*` `/`

You can apply the usual arithmetic operators, such as `+`, `-`, `*` and `/`, to intervals.

The result of adding two intervals is the interval whose endpoints are the sums of the left end points and the right end points.

Example.

Input:

```
i[1,4] + i[2,3]
```

Output:

```
[3.0000000000000000..7.0000000000000000]
```

The negative of an interval is the result of taking the negative of the end points of the interval. The new end points will have to be switched.

Example.

Input:

```
-i[2,3]
```


Output:

$[-3.0000000000000000.. - 2.0000000000000000]$

The product of two intervals is the interval whose endpoints are the product of the left endpoints of the two intervals and the product of the right endpoints of the two intervals. The smallest product will be the left endpoint of the product interval, and the largest product will be the right endpoint of the product interval.

Examples.

- *Input:*

$i[1,4]*i[2,3]$

Output:

$[2.0000000000000000..0.1200000000000000e2]$

- *Input:*

$i[-2,4]*i[3,5]$

Output:

$[-0.1000000000000000e2..0.2000000000000000e2]$

The reciprocal of an interval is the interval determined by the reciprocals of the end points.

Examples.

- *textitInput:*

$1/i[2,3]$

Output:

$[0.3333333333333333..0.5000000000000000]$

- *Input:*

$1/i[-6,-3]$

Output:

$[-0.3333333333333333.. - 0.1666666666666667]$

If the original interval has zero as an end point, then the reciprocal interval will have plus or minus infinity as one of the end points. If one end point is positive and the other is negative, then the reciprocal will simply be the interval from -infinity to infinity.

Examples.

- *Input:*

$$1/i[0,2]$$

Output:

$$[0.5000000000000000.. + \infty]$$

- *Input:*

$$1/i[-1,0]$$

Output:

$$[-\infty.. -1.0000000000000000]$$

Input:

$$1/i[-2,3]$$

Output:

$$[-\infty.. + \infty]$$

You can also, if you want, do the usual operations such as subtraction, division, powers and roots.

6.38.4 The midpoint of an interval: `midpoint`

The `midpoint` operator finds the midpoint of an interval.

- `midpoint` takes one argument:
 I , an interval.
- `midpoint(I)` returns the midpoint of I .

Example.

Input:

$$\text{midpoint}(i[2,3])$$

Output:

$$2.5000000000000000$$

6.38.5 The union of intervals: `union`

In Xcas, the union of two intervals is their convex hull.

The `union` operator is a binary infix operator that can find the union of two intervals.

Examples.

- *Input:*

$$i[1,3] \text{ union } i[2,4]$$

Output:

$$[1.0000000000000000..4.0000000000000000]$$

- *Input:*

```
i[2,4] union i[6,9]
```

Output:

```
[2.0000000000000000..9.0000000000000000]
```

6.38.6 The intersection of intervals: `intersect`

The `intersect` operator is a binary infix operator that finds the intersection of two intervals.

Example.

Input:

```
i[1,3] intersect i[2,4]
```

Output:

```
[2.0000000000000000..3.0000000000000000]
```

6.38.7 Testing if an object is in an interval: `contains`

The `contains` command determines if an object is in an interval.

- `contains` takes two arguments:
 - I , an interval.
 - obj , an object.
- `contains(I, obj)` returns 1 if obj is in I ; i.e., either obj is a number which is contained in I or obj is an interval which is a subset of I . It returns 0 otherwise.

Examples.

- *Input:*

```
contains(i[0,2],1)
```

Output:

```
1
```

- *Input:*

```
contains(i[0,2],3)
```

Output:

```
0
```

- *Input:*

```
contains(i[0,2],i[1,2])
```

Output:

```
1
```

6.38.8 Converting a number into an interval: `convert`

The `convert` command (see Section 6.23.26 p.323) can convert an expression which evaluates to a number to the smallest interval which contains the number.

- `convert` takes two mandatory arguments and one optional argument:
 - *expr*, a number which evaluates to a number.
 - *interval*, a reserved word.
 - Optionally, *n*, an integer greater than 15 giving the desired number of digits.
- `convert(expr, interval <, n>)` returns the smallest interval containing the value of *expr*.

Examples.

- *Input:*

```
convert(sin(3)+1, interval)
```

Output:

```
[1.14112000805985..1.14112000805990]
```

- *Input:*

```
convert(sin(3)+1, interval, 20)
```

Output:

```
[1.1411200080598672220..1.1411200080598672222]
```

6.38.9 Converting box constraints from matrix to interval form: `box_constraints`

The `box_constraints` command can convert bounds of given variables from the matrix form to interval form.

- `box_constraints` takes two mandatory arguments:
 - *vars*, a list of variables $[x_1, x_2, \dots, x_n]$.
 - *bnds*, a matrix with *n* rows and two columns in which the *k*-th row contains the lower and upper bound l_k and u_k of x_k , respectively.
- `box_constraints(vars, bnds)` returns the sequence $(x_1 = l_1 \dots u_1, x_2 = l_2 \dots u_2, \dots, x_n = l_n \dots u_n)$.

Examples.

- *Input:*

```
box_constraints([x,y], [[-2,2], [-5,5]])
```

Output:

```
 $x = -2 \dots 2, y = -5 \dots 5$ 
```

6.39 Sequences and lists

6.39.1 Defining a sequence or a list: `seq[] ()`

Recall (see Section 5.3.1 p.97) that a sequence is represented by a sequence of elements separated by commas, either without delimiters, with parentheses ((and)) as delimiters, or with `seq[` and `]` as delimiters.

Examples.

Input:

`a,b,c,d`

or:

`(a,b,c,d)`

or:

`seq[a,b,c,d]`

Output:

`a,b,c,d`

Similarly (see Section 5.3.3 p.98) a list (or a vector) is a sequence of elements separated by commas delimited with `[` and `]`.

Examples.

- *Input:*

`[1,2,5]`

Output:

`[1,2,5]`

- *Input:*

`[]`

(to create the empty list).

Output:

`[]`

Lists have more structure than sequences. For example, a list can contain lists (for example, a matrix is a list of lists of the same size, see Section 6.44 p.502). Lists may be used to represent vectors (lists of coordinates), matrices, or univariate polynomials (lists of coefficients by decreasing order, see Section 6.27.1 p.351).

Sequences, on the other hand, are flat. An element of a sequence cannot be a sequence.

See Section 6.40 p.463 for some commands only for lists.

6.39.2 Making a sequence or a list: `seq` `$`

The `seq` command or `$` operator can create a sequence or a list.

To create a sequence:

- `seq` takes three mandatory arguments and one optional argument:
 - *expr*, an expression depending on a parameter.
 - *k*, the parameter.
 - *a..b*, a range of values. The range can be combined with the parameter into one argument of *k = a..b*.
 - Optionally *p*, a step size (by default 1 or -1, depending on whether $b > a$ or $b < a$). This is only allowed if the previous two arguments are combined into one, *k = a..b*

This is Maple-like syntax.

- `seq(expr, k, a..b)` (or `seq(expr, k=a..b⟨, p⟩)`) returns the sequence formed by the values of *expr*, as *k* changes from *a* to *b* in steps of *p*.

Alternatively, a sequence can be created with the infix `$` operator. Namely, *expr*`$ k=a..b` returns the sequence formed by the values of *expr* as *k* changes from *a* to *b*. As a special case, *expr*`$ n` creates a sequence consisting of *n* copies of *expr*.

There are two ways to create a list with `seq`.

First:

- `seq` takes four mandatory arguments and one optional argument:
 - *expr*, an expression depending on a parameter.
 - *k*, the parameter.
 - *a*, the beginning value of the parameter.
 - *b*, the ending value of the parameter.
 - Optionally *p*, a step size (by default 1 or -1, depending on whether $b > a$ or $b < a$).

This is TI-like syntax.

- `seq(expr, k, a, b⟨, p⟩)` returns the list consisting of the values of *expr*, as *k* changes from *a* to *b* in steps of *p*.
- As a special case, `seq(expr, n)` creates a list consisting of *n* copies of *expr*.

Second:

- `seq` takes two arguments: argument:
 - *expr*, an expression.
 - *n*, a positive integer.

- `seq(expr, n)` returns the list consisting of n copies of `expr`.

Remark.

- In `Xcas` mode, the precedence of `$` is not the same as it is, for example, in `Maple`. In case of doubt, put the arguments of `$` in parenthesis. For example, the equivalent of `seq(j^2, j=-1..3)` is `(j^2)$(j=-1..3)` and returns `(1,0,1,4,9)`.
- With `Maple` syntax, `j, a..b, p` is not valid. To specify a step p for the variation of j from a to b , use `j=a..b, p` or use the TI syntax `j, a, b, p` and get the sequence from the list with `op(...)`.

Examples.

- To create a sequence:

Input:`seq(j^3, j, 1..4)`*or:*`seq(j^3, j=1..4)`*or:*`(j^3)$(j=1..4)`*Output:*`1, 8, 27, 64`

- To create a list:

Input:`seq(j^3, j, 1, 4)`*Output:*`[1, 8, 27, 64]`

- To create a sequence:

Input:`seq(j^3, j=-1..4, 2)`*Output:*`-1, 1, 27`

To create a list:

Input:`seq(j^3, j, -1, 4, 2)`

Output:

$$[-1, 1, 27]$$

- *Input:*

$$\text{seq}(j^3, j, 0, 5, 2)$$

Output:

$$[0, 8, 64]$$

- *Input:*

$$\text{seq}(j^3, j, 5, 0, -2)$$

or:

$$\text{seq}(j^3, j, 5, 0, 2)$$

Output:

$$[125, 27, 1]$$

- *Input:*

$$\text{seq}(j^3, j, 1, 3, 0.5)$$

Output:

$$[1, 3.375, 8.0, 15.625, 27.0]$$

- *Input:*

$$\text{seq}(j^3, j, 1, 3, 1/2)$$

Output:

$$\left[1, \frac{27}{8}, 8, \frac{125}{8}, 27\right]$$

- To create a list with several copies of the same element:

Input:

$$\text{seq}(t, 4)$$

Output:

$$[t, t, t, t]$$

- To create a sequence with several copies of the same element:

Input:

$$\text{seq}(t, k=1..4)$$

or:

$$t\$4$$

Output:

$$t, t, t, t$$

Examples of sequences being used.

- Find the third derivative of $\ln(t)$:
(See Section 6.19.4 p.271).)

Input:

$$\text{diff}(\log(t), t\$3)$$

Output:

$$\frac{2}{t^3}$$

- *Input:*

```
l:=[ [2,3] , [5,1] , [7,2] ]
seq((l[k][0])$(l[k][1]),k=0 .. size(l)-1)
```

Output:

$$(2, 2, 2), (5), (7, 7)$$

then: *Input:*

$$\text{eval}(\text{ans}())$$

Output:

$$2, 2, 2, 5, 7, 7$$

- Transform a string into a list of its characters:

Input:

```
chn := "abracadbra"
seq(chn[j],j,0,size(chn)-1)
```

Output:

$$["a", "b", "r", "a", "c", "a", "d", "a", "b", "r", "a"]$$

6.39.3 Length of a sequence or list: size nops length

You can find the length of a sequence or list with any of the **size**, **nops** or **length** commands.

- **size** (or **nops** or **length**) takes one argument:
 S , a sequence or list.
- **size**(S) (or **nops**(S) or **length**(S)) returns the length of L .

Examples.

- *Input:*

`nops(a,e,i,o,u)`

or:

`size(a,e,i,o,u)`

or:

`length(a,e,i,o,u)`

Output:

5

- *Input:*

`nops([3,4,2])`

or:

`size([3,4,2])`

or:

`length([3,4,2])`

Output:

3

6.39.4 Getting the first element of a sequence or list: head

The head command finds the first element of a sequence or list.

- `head` takes one argument:
 S , a sequence or list.
- `head(S)` returns the first element of S .

Examples.

- *Input:*

`head(A,B,C,D)`

Output:

A

- *Input:*

`head([0,1,2,3])`

Output:

0

6.39.5 Getting a sequence or list without the first element: `tail`

The `tail` command removes the first element of a list.

- `tail` takes one argument:
 L , a list.
- `tail(L)` returns L without its first element.

Example.

Input:

```
tail([0,1,2,3])
```

Output:

```
[1,2,3]
```

6.39.6 Getting an element of a sequence or a list: `[] [][]` at

The elements of a sequence have indices beginning at 0 in `Xcas` mode and 1 in other modes (see Section 3.5.2 p.71).

You can get the an element of index n of a sequence or list by following the sequence or list with `[n]` (see Section 5.3.4 p.99). (Note that `head(S)` does the same thing as `S[0]`.)

Examples.

- *Input:*

```
(0,3,2)[1]
```

Output:

```
3
```

- *Input:*

```
S := 2,3,4,5
S[2]
```

Output:

```
4
```

- *Input:*

```
[A,B,C,D][2]
```

Output:

```
C
```

For lists, the `at` command can also be used to get the element at a specific position.

- **at** takes two arguments:
 - L , a list.
 - n , an integer.
- **at**(L, n) returns the element of S with index n .

Note. **at** cannot be used for sequences, since the second argument would be merged with the sequence.

Example.

Input:

$$[0, 1, 2][1]$$

or:

$$\text{at}([0, 1, 2], 1)$$

Output:

$$1$$

6.39.7 Finding a subsequence or a sublist

The bracket notation used to find elements of sequences and lists can also be used to extract a range of elements. If S is a sequence or list of size n , then $S[n_1..n_2]$ returns the subsequence or sublist of S consisting of the elements with indices from n_1 to n_2 , where $0 \leq n_1 \leq n_2 < s$ (in Xcas syntax mode) or $0 < n_1 \leq n_2 \leq s$ in other syntax modes.

Examples.

- *Input:*

$$[0, 1, 2, 3, 4][1..3]$$

Output:

$$[1, 2, 3]$$

- *Input:*

$$(A, B, C, D, E)[1..3]$$

Output:

$$B, C, D$$

For lists, the **at** command can also be used to get a sublist.

- **at** takes two arguments:
 - L , a list.
 - $n_1..n_2$, a range of integers.

- `at(L, n1..n2)` returns the sublist of L consisting of the elements with indices from n_1 to n_2 .

Again, `at` can not be used for sequences.

Example.

Input:

```
at([1,2,3,4,5],2..4
```

Output:

```
[3,4,5]
```

An alternative to using `at` for finding a sublist is the `mid` command, which again cannot be used for sequences.

- `mid` takes two mandatory and one optional argument:
 - L , a list.
 - n , the index of the beginning of the sublist.
 - Optionally, l , the length of the sublist.
- `mid(L, n⟨, l⟩)` returns the sublist of L with index beginning at n . With the option l , the length of the sublist will be l , otherwise it will go to the end of the list L .

Examples.

- *Input:*

```
mid([0,1,2,3,4,5],2,3)
```

Output:

```
[2,3,4]
```

- *Input:*

```
mid([0,1,2,3,4,5],2)
```

Output:

```
[2,3,4,5]
```

6.39.8 Concatenating two sequences: ,

The `,` operator is an infix operator which concatenates two sequences. (Note that it does not concatenate lists.)

Example.

Input:

```
A:=(1,2,3,4)
B:=(5,6,3,4)
A,B
```

Output:

```
1,2,3,4,5,6,3,4
```

6.39.9 The + operator applied on sequences and lists

The infix operator `+`, with two sequences as arguments, returns the total sum of the elements of the two sequences. This is different than with two lists as arguments, where the term by term sums of the elements of the two lists would be returned. (See Section 6.16.1 p.239.) To use it as a prefix, it has to be quoted (`'+'`).

Examples.

- *Input:*

$$(1,2,3,4,5,6)+(4,3,5)$$

or:

$$'+'((1,2,3,4,5,6),(4,3,5))$$

Output:

$$33$$

- *Input:*

$$[1,2,3,4,5,6]+[4,3,5]$$

or:

$$'+'([1,2,3,4,5,6],[4,3,5])$$

Output:

$$[5,5,8,4,5,6]$$

6.39.10 Transforming sequences into lists and lists into sequences: `[]` `nop` `op` `makesuite`

To transform a sequence into list, you can put square brackets (`[]`) around the sequence. The `makevector` and `nop` commands have the same effect.

- `makevector` (or `nop`) takes one argument:
 S , a sequence.
- `makevector(S)` (or `nop(S)`) returns the list with the same elements as S in the same order.

Example.

Input:

$$[\text{seq}(j^3, j=1..4)]$$

or:

$$[(j^3)\$(j=1..4)]$$

or:

```
nop(j^3$(j=1..4))
```

or:

```
makevector(j^3$(j=1..4))
```

Output:

```
[1, 8, 27, 64]
```

The `makesuite` command transforms a list into a sequence. Note that `op` (see Section 6.15.3 p.236) can do the same thing.

- `makesuite` takes one argument:
 L , a list.
- `makesuite(L)` returns the sequence with the same elements as L and the same order.

Example.

Input:

```
makesuite([0,1,2])
```

or:

```
op([0,1,2])
```

Output:

```
0, 1, 2
```

6.40 Operations on lists

6.40.1 Sizes of a list of lists: `sizes`

Recall that one thing that distinguishes lists from sequences is that a list can be an element of another list. The `sizes` command finds the length of elements of a list, as long as each element is a list.

- `sizes` takes one argument:
 L , a list each of whose elements is a list.
- `sizes(L)` returns the list of the lengths of the elements of L .

Example.

Input:

```
sizes([[3,4],[2]])
```

Output:

```
[2, 1]
```

6.40.2 Making a list with a function: makelist

The `makelist` command creates lists built from values of a function.

- `makelist` takes three mandatory arguments and one optional argument:
 - f , a function (see Section 6.15.1 p.234).
 - a and b , two real numbers.
 - Optionally p , a step size (by default 1 if $b > a$ and -1 if $b < a$).
- `makelist(f, a, b, p)` returns the list $[f(a), f(a + p), \dots, f(a + kp)]$ with k such that: $a < a + kp \leq b < a + (k + 1)p$ or $a > a + kp \geq b > a + (k + 1)p$.

Examples.

(In these examples, purge `x` if `x` is not symbolic.)

- *Input:*

```
makelist(x->x^2,3,5)
```

or:

```
makelist(x->x^2,3,5,1)
```

or: Input:

```
h(x):=x^2
makelist(h,3,5,1)
```

Output:

```
[9, 16, 25]
```

- *Input:*

```
makelist(x->x^2,3,6,2)
```

Output:

```
[9, 25]
```

- *Input:*

```
makelist(4,1,3)
```

regards 4 as the constant function, and so creates a list with entries 4, from integers 1 to 3. This is the same as `[4 $ 3]`.

Output:

```
[4, 4, 4]
```


6.40.3 Making a list with zeros: `newList`

The `newList` makes a list of all zeros.

- `newList` takes one argument:
 n , a positive integer.
- `newList(n)` returns a list of n zeros.

Example.

Input:

```
newList(3)
```

Output:

```
[0, 0, 0]
```

6.40.4 Making a list of integers: `range`

The `range` command creates lists of equally spaced numbers. It can take one, two or three arguments.

With one argument:

- `range` takes one argument:
 n , a positive integer.
- `range(n)` returns the list $[0, 1, \dots, n - 1]$.

Example.

Input:

```
range(5)
```

Output:

```
[0, 1, 2, 3, 4]
```

With two or three arguments:

- `range` takes two mandatory and one optional argument:
 - a and b , two real numbers with $a < b$ (unless the third argument p is provided and negative).
 - Optionally, p , a nonzero real number used for the step size (by default 1).
(If $p < 0$, then a must be larger than b .)
- `range(a, b [p])` returns the list $[a, a + p, \dots]$ up to, but not including, b .

Examples.

- *Input:*

```
range(4, 10)
```

Output:

[4, 5, 6, 7, 8, 9]

- *Input:*

`range(2.3, 7.4)`

Output:

[2.3, 3.3, 4.3, 5.3, 6.3, 7.3]

- *Input:*

`range(4, 13, 2)`

Output:

[4, 6, 8, 10, 12]

- *Input:*

`range(10, 4, -1)`

Output:

[10, 9, 8, 7, 6, 5]

You can use the **range** command to create a list of values $f(k)$, where k is an integer satisfying a certain condition. (See Section 12.3.3 p.851 for the **for** loop used below.)

- You can list the values of an expression in a variable which goes over a range defined by **range**.

Input:

`[k^2 + k for k in range(10)]`

Output:

[0, 2, 6, 12, 20, 30, 42, 56, 72, 90]

- You can list the values of an expression in a variable which goes over a range defined by **range** and which satisfies a given condition.

Input:

`[k for k in range (4,10) if isprime(k)]`

(See Section 6.5.14 p.145 for **isprime**.)

Output:

[5, 7]

Input:

`[k^2 + k for k in range(1,10,2) if isprime(k)]`

Output:

[12, 30, 56]

6.40.5 Selecting elements of a list: `select`

The `select` command selects elements of a list meeting given conditions.

- `select` takes two arguments:
 - f , a boolean function.
 - L , a list.
- `select(f, L)` returns the sublist of L consisting of the elements c such that $f(c) == \text{true}$.

Example.

Input:

```
select(x->(x>=2), [0,1,2,3,1,5])
```

Output:

```
[2,3,5]
```

6.40.6 The left and right portions of a list: `left` `right`

The `left` and `right` can find the left and right parts of a list. (See Section 6.3.4 p.123, Section 6.15.3 p.236, Section 6.37.1 p.445, Section 6.38.2 p.448, Section 6.55.4 p.618 and Section 6.55.5 p.618 for other uses of `left` and `right`.)

- `left` takes two arguments:
 - L , a list.
 - n , a positive integer.
- `left(L, n)` returns the first n elements of L .

Example.

Input:

```
left([0,1,2,3,4,5,6,7,8], 3)
```

Output:

```
[0,1,2]
```

- `right` takes two arguments:
 - L , a list.
 - n , a positive integer.
- `right(L, n)` returns the last n elements of L .

Example.

Input:

```
right([0,1,2,3,4,5,6,7,8], 4)
```

Output:

```
[5,6,7,8]
```

6.40.7 Modifying the elements of a list: subsop

The `subsop` command can be used to modify elements in a list.

- `subsop` takes two arguments:
 - L , a list.
 - $i=value$, an index and a new value.
- (In all but `Maple` mode).
`subsop(L, i=value)` returns the list L with the value at index i replaced by $value$.
- (In `Maple` mode; the only difference is the order of the arguments).
`subsop(i=value, L)` returns the list L with the value at index i replaced by $value$.

Remark: If the second argument is $i=NULL$, then the element at index i is removed from L .

You can also redefine elements (or define new elements, but not remove elements) with `:=`.

Examples.

- *Input (in Xcas mode, the index of the first element is 0):*

```
subsop([0,1,2],1=5)
```

or:

```
L:=[0,1,2];L[1]:=5
```

Output:

```
[0,5,2]
```

- *Input (in Xcas mode, the index of the first element is 0):*

```
subsop([0,1,2], '1=NULL')
```

Output:

```
[0,2]
```

- *Input (in Mupad TI mode, the index of the first element is 1):*

```
subsop([0,1,2],2=5)
```

or:

```
L:=[0,1,2];L[2]:=5
```

Output:

```
[0,5,2]
```

- When using `:=` to insert an element in a list, the list will be padded with 0s if necessary.

Input:

$$L := []$$

then:

$$L[3] := 5$$

Output:

$$[0, 0, 0, 5]$$

- In **Maple** mode the arguments are permuted and the index of the first element is 1.

Input:

$$\text{subsop}(2=5, [0, 1, 2])$$

or:

$$L := [0, 1, 2]; L[2] := 5$$

Output:

$$[0, 5, 2]$$

6.40.8 Removing an element in a list: `suppress`

The `suppress` command removes elements from a list.

- `suppress` takes two arguments:
 - L , a list.
 - i , a nonnegative integer.
- `suppress(L, i)` returns the list L with the element at index i removed.

Example.

Input:

$$\text{suppress}([3, 4, 2], 1)$$

Output:

$$[3, 2]$$

6.40.9 Removing elements of a list: `remove`

The `remove` command removes elements of a list according to a given conditions.

- `remove` takes two arguments:
 - f , a boolean function.
 - L , a list.
- `remove(f, L)` returns the sublist of L with the elements c such that $f(c) == \text{true}$ removed.

Example.

Input:

```
remove(x->(x>=2), [0,1,2,3,1,5])
```

Output:

```
[0,1,1]
```

Remark: You can use `remove` to remove characters from a string. For example, to remove all the "a"s of a string (see Section 12.1.2 p.835 for writing functions):

Input:

```
orda := ord("a");;
```

then:

```
f(chn):={
  local l:=length(chn)-1;
  return remove(x->(ord(x)==orda), seq(chn[k], k, 0, l));
}
```

Now:

Input:

```
f("abracadabra")
```

Output:

```
["b", "r", "c", "d", "b", "r"]
```

To get a string:

Input:

```
char(ord(["b", "r", "c", "d", "b", "r"]))
```

Output:

```
"brcdbr"
```

6.40.10 Inserting an element into a list or a string: insert

The `insert` command inserts elements into a list or string.

- `insert` takes three arguments:
 - L , a list or a string.
 - i , an integer (the index).
 - x .
- `insert(L, i, x)` returns L with x inserted at index i and the necessary elements shifted to the right.

Examples.

- *Input:*

```
insert([3,4,2],2,5)
```

Output:

```
[3,4,5,2]
```

- *Input:*

```
insert("342",2,"5")
```

Output:

```
"3452"
```

`insert` returns an error if the index is too large.

Input:

```
insert([3,4,2],4,5)
```

Output:

```
insert([3,4,2],4,5)  
Error: Invalid dimension
```

6.40.11 Appending an element at the end of a list: append

The `append` command adds an element to the end of a list.

- `append` takes two arguments:
 - L , a list.
 - x .
- `append(L, x)` returns a list L with the additional element x at the end.

Examples.

- *Input:*

```
append([3,4,2],1)
```

Output:

```
[3,4,2,1]
```

- *Input:*

```
append([1,2],[3,4])
```

Output:

```
[1,2,[3,4]]
```

6.40.12 Prepending an element at the beginning of a list: prepend

The **prepend** command adds an element to the beginning of a list.

- **prepend** takes two arguments:
 - x .
 - L , a list.
- **prepend**(x, L) returns a list containing x as the first element followed by the elements of L .

Examples.

- *Input:*

```
prepend([3,4,2],1)
```

Output:

```
[1,3,4,2]
```

- *Input:*

```
prepend([1,2],[3,4])
```

Output:

```
[[3,4],1,2]
```

6.40.13 Concatenating two lists or a list and an element: concat augment

The **concat** command combines two lists or adds an element to a list. **augment** is a synonym for **concat**.

- **concat** takes two arguments:
 - L_1 and L_2 , two lists or a list and an element (in any order).
- **concat**(L_1, L_2) returns a list consisting of the elements of L_1 (or L_1 itself if it isn't a list) followed by the elements of L_2 (or L_2 itself).

Examples.

- *Input:*

$$\text{concat}([3, 4, 2], [1, 2, 4])$$

or:

$$\text{augment}([3, 4, 2], [1, 2, 4])$$

Output:

$$[3, 4, 2, 1, 2, 4]$$

- *Input:*

$$\text{concat}([3, 4, 2], 5)$$

or:

$$\text{augment}([3, 4, 2], 5)$$

Output:

$$[3, 4, 2, 5]$$

- *Input:*

$$\text{concat}(2, [5, 4, 3])$$

or:

$$\text{augment}(2, [5, 4, 3])$$

Output:

$$[2, 5, 4, 3]$$
Warning.

Input:

$$\text{concat}([[3, 4, 2]], [[1, 2, 4]])$$

or:

$$\text{augment}([[3, 4, 2]], [[1, 2, 4]])$$

results in:

Output:

$$[\ 3 \ 4 \ 2 \ 1 \ 2 \ 4 \]$$

6.40.14 Flattening a list: `flatten`

The `flatten` command replaces any sublists of a list by their elements.

- `flatten` takes one argument:
 L , a list.
- `flatten(L)` returns a list which is the result of recursively replacing any elements that are lists by the elements, resulting in a list with no lists as elements.

Example.

Input:

```
flatten([[1, [2, 3], 4], [5, 6]])
```

Output:

```
[1, 2, 3, 4, 5, 6]
```

If the original list is a matrix, you can also use the `mat2list` command for this (see Section 6.40.33 p.489).

6.40.15 Reversing order in a list: `revlist`

The `revlist` command reverses the elements of a list or sequence.

- `revlist` takes one argument:
 L , a list or sequence.
- `revlist(L)` returns a list or sequence with the same elements as L in the reverse order.

Examples.

- *Input:*

```
revlist([0, 1, 2, 3, 4])
```

Output:

```
[4, 3, 2, 1, 0]
```

- *Input:*

```
revlist([0, 1, 2, 3, 4], 3)
```

Output:

```
3, [0, 1, 2, 3, 4]
```

6.40.16 Rotating a list: rotate

The `rotate` command rotates a list; namely it moves elements from one side and puts them on the other.

- `rotate` takes one mandatory argument and one optional argument:
 - L , a list.
 - Optionally, n , an integer (by default $n = -1$).
- `rotate(L)` returns the list formed by rotating L n places to the left if $n > 0$ or $-n$ places to the right if $n < 0$. Elements leaving the list from one side come back on the other side. By default $n = -1$ and the last element becomes the first.

Examples.

- *Input:*

```
rotate([0,1,2,3,4])
```

Output:

```
[4, 0, 1, 2, 3]
```

- *Input:*

```
rotate([0,1,2,3,4],2)
```

Output:

```
[2, 3, 4, 0, 1]
```

- *Input:*

```
rotate([0,1,2,3,4],-2)
```

Output:

```
[3, 4, 0, 1, 2]
```

6.40.17 Shifting the elements of a list: shift

The `shift` command shifts the elements of a list.

- `shift` takes one mandatory argument and one optional argument:
 - L , a list.
 - Optionally, n , an integer (by default $n = -1$).
- `shift(L)` returns the list formed by shifting the elements of L n places to the left if $n > 0$ or $-n$ places to the right if $n < 0$. Elements leaving the list from one side are replaced by 0s on the other side.

Examples.

- *Input:*

`shift([0,1,2,3,4])`

Output:

`[0,0,1,2,3]`

- *Input:*

`shift([0,1,2,3,4],2)`

Output:

`[2,3,4,0,0]`

- *Input:*

`shift([0,1,2,3,4],-2)`

Output:

`[0,0,0,1,2]`

6.40.18 Sorting: `sort`

The `sort` command sorts lists and expression in various ways.

- `sort` takes one mandatory argument and one optional argument:
 - L , a list or expression.
 - Optionally, f , a boolean function of two variables (by default, f if the function $(x,y) \rightarrow x \leq y$).
- `sort(L, f)` (for a list L) returns a copy of L sorted according to the order given by f . By default, this means that it will be sorted in increasing order.
- `sort(L, f)` (for an expression L) returns a copy of L with the terms in sums and products collected and sorted.

Note that using $(x,y) \rightarrow x \geq y$ for f will to sort the list in decreasing order. This may also be used to sort a list of lists (that `sort` with one argument does not know how to sort).

Examples.

- *Input:*

`sort([3,4,2])`

Output:

`[2,3,4]`

- *Input:*

```
sort(exp(2*ln(x))+x*y-x+y*x+2*x)
```

Output:

$$2xy + e^{2\ln x} + x$$

Input:

```
simplify(exp(2*ln(x))+x*y-x+y*x+2*x)
```

Output:

$$x^2 + 2xy + x$$

- *Input:*

```
sort([3,4,2],(x,y)->x>=y)
```

Output:

$$[4, 3, 2]$$

6.40.19 Sorting a list by increasing order: SortA

The **SortA** command sorts a list or a matrix in increasing order.

- **SortA** takes one argument:
 L , a list.
- If L is not a matrix:
SortA(L) returns a copy of L with the elements in increasing order (if L is not a matrix).
 If L is a matrix:
SortA(L) returns a copy of L where the columns are sorted according to increasing order in the first row.

Examples.

- *Input:*

```
SortA([3,4,2])
```

Output:

$$[2, 3, 4]$$

- *Input :*

```
SortA([[3,4,2],[6,4,5]])
```

Output:

$$\begin{bmatrix} 2 & 3 & 4 \\ 5 & 6 & 4 \end{bmatrix}$$

6.40.20 Sorting a list by decreasing order: SortD

The `SortD` command sorts a list or a matrix in decreasing order.

- `SortD` takes one argument:
 L , a list.
- If L is not a matrix:
`SortA(L)` returns a copy of L with the elements in decreasing order.
 If L is a matrix:
`SortA(L)` returns a copy of L where the columns are sorted according to decreasing order in the first row.

Examples.

- *Input:*

$$\text{SortD}([3,4,2])$$

Output:

$$[4,3,2]$$

- *Input:*

$$\text{SortD}([[3,4,2], [6,4,5]])$$

Output:

$$\begin{bmatrix} 4 & 3 & 2 \\ 4 & 6 & 5 \end{bmatrix}$$
6.40.21 Number of elements equal to a given value: count_eq

The `count_eq` command counts the number of elements of a list equal to a given value.

- `count_eq` takes two arguments:
 - x , a real number.
 - L , a list or matrix of real numbers.
- `count_eq(x, L)` returns the number of elements of L which are equal to x .

Example.

Input:

$$\text{count_eq}(12, [2,12,45,3,7,78])$$

Output:

6.40.22 Number of elements smaller than a given value: `count_inf`

The `count_inf` command counts the number of elements of a list strictly less than a given value.

- `count_inf` takes two arguments:
 - x , a real number.
 - L , a list or matrix of real numbers.
- `count_inf(x, L)` returns the number of elements of L which are strictly less than x .

Example.

Input:

```
count_inf(12, [2, 12, 45, 3, 7, 78])
```

Output:

3

6.40.23 Number of elements greater than a given value: `count_sup`

The `count_sup` command counts the number of elements of a list strictly greater than a given value.

- `count_sup` takes two arguments:
 - x , a real number.
 - L , a list or matrix of real numbers.
- `count_sup(x, L)` returns the number of elements of L which are strictly greater than x .

Example.

Input:

```
count_sup(12, [2, 12, 45, 3, 7, 78])
```

Output:

2

6.40.24 Sum of elements of a list: `sum` `add`

The `sum` command adds the elements of a list or sequence of numbers. `add` is a synonym for `sum`.

- `sum` takes one argument:
 - L , a list or sequence of numbers.
- `sum(L)` returns the sum of the elements of L .

Example.*Input:*

$$\text{sum}(2, 3, 4, 5, 6)$$
Output:

$$20$$
6.40.25 Sum of list (or matrix) elements transformed by a function: count

The `count` command adds the values of a function applied to the elements of a list or matrix.

- `count` takes two arguments:
 - f , a real-valued or boolean-valued function.
 - L , a list or a matrix.
- `count(f, L)` returns the sum of $f(x)$ for all elements x in L .
 If f is a boolean-valued function, this is just the number of elements for which the boolean is true.

Examples.

- *Input:*

$$\text{count}((x) \rightarrow x, [2, 12, 45, 3, 7, 78])$$
Output:

$$147$$

because: $2 + 12 + 45 + 3 + 7 + 78 = 147$.

- *Input:*

$$\text{count}((x) \rightarrow x < 12, [2, 12, 45, 3, 7, 78])$$
Output:

$$3$$

- *Input:*

$$\text{count}((x) \rightarrow x == 12, [2, 12, 45, 3, 7, 78])$$
Output:

$$1$$

- *Input:*

$$\text{count}((x) \rightarrow x > 12, [2, 12, 45, 3, 7, 78])$$

Output:

2

- *Input:*

```
count(x->x^2,[3,5,1])
```

Output:

35

Indeed $3^2 + 5^2 + 1^1 = 35$.

- *Input:*

```
count(id,[3,5,1])
```

Output:

9

Indeed, `id` is the identity functions and $3+5+1=9$.

- *Input:*

```
count(1,[3,5,1])
```

Output:

3

Indeed, 1 is the constant function equal to 1 and $1+1+1=3$.

6.40.26 Cumulated sum of the elements of a list: `cumSum`

The `cumSum` command finds the partial sums of a list or sequence; namely, the k th element of the

- `cumSum` takes one argument:
 L , a list or sequence of numbers or of strings.
- `cumSum(L)` returns the list or sequence with same length as L and whose k th element is the sum or concatenation of elements 0 through k of L .

Examples.

- *Input:*

```
cumSum(sqrt(2),3,4,5,6)
```

Output:

$\sqrt{2}, \sqrt{2} + 3, \sqrt{2} + 3 + 4, \sqrt{2} + 3 + 4 + 5, \sqrt{2} + 3 + 4 + 5 + 6$

Input:

```
normal(cumSum(sqrt(2),3,4,5,6))
```

Output:

$$\sqrt{2}, \sqrt{2} + 3, \sqrt{2} + 7, \sqrt{2} + 12, \sqrt{2} + 18$$

• *Input:*

```
cumSum(1.2,3,4.5,6)
```

Output:

$$1.2, 4.2, 8.7, 14.7$$

• *Input:*

```
cumSum([0,1,2,3,4])
```

Output:

$$[0, 1, 3, 6, 10]$$

• *Input:*

```
cumSum("a","b","c","d")
```

Output:

$$"a", "ab", "abc", "abcd"$$

• *Input:*

```
cumSum(["a","ab","abc","abcd"])
```

Output:

$$["a", "aab", "aababc", "aababcabcd"]$$

6.40.27 Product: product mul

The **product** command can find the products of elements of an expression (see Section 6.40.27 p.482), the elements of a list (see Section 6.40.27 p.483), the elements of the columns of a matrix (see Section 6.45.6 p.528), and the term-by-term (Hadamard) product of two matrices (see Section 6.45.8 p.529).

Product of values of an expression: product

The **product** command can find the product of the values of an expression as the variable changes.

Here, **mul** is a synonym for **product**.

- **product** takes four mandatory arguments and one optional argument:
 - *expr*, an expression.
 - *x*, the name of a variable.

- a and b , real numbers.
- Optionally p , a real number representing a step size. (By default $p = 1$).
- `product(expr x, a, b <p>)` returns the product of the values of `expr` as x goes from a to b in steps of p .
This syntax is for compatibility with Maple.

Examples.

- *Input:*

```
product(x^2+1,x,1,4)
```

or:

```
mul(x^2+1,x,1,4)
```

Output:

1700

Indeed $(1^2 + 1) \cdot (2^2 + 1) \cdot (3^2 + 1) \cdot (4^2 + 1) = 1700$

- *Input:*

```
product(x^2+1,x,1,5,2)
```

or:

```
mul(x^2+1,x,1,5,2)
```

Output:

520

Indeed $(1^2 + 1) \cdot (3^2 + 1) \cdot (5^2 + 1) = 520$

Product of elements of a list: product

The `product` command can find the products of elements of a list. For this, `mul` is a synonym for `product`.

- `product` takes one mandatory and one optional argument:
 - L , a list of numbers.
 - Optionally, L_2 , another list of numbers the same length as L .
- `product(L)` returns the product of the elements of L .
- `product(L, L2)` returns the term by term product of the elements of L and L_2 .

(See also Section [6.45.8](#) p.529.)

Examples.

- *Input:*

```
product([2,3,4])
```

or:

```
mul([2,3,4])
```

Output:

```
24
```

- *Input:*

```
product([[2,3,4],[5,6,7]])
```

Output:

```
[10,18,28]
```

- *Input:*

```
product([2,3,4],[5,6,7])
```

or:

```
mul([2,3,4],[5,6,7])
```

Output:

```
[10,18,28]
```

6.40.28 Applying a function of one variable to the elements of a list: `map` `apply`

The `apply` and `map` commands can both apply a function to a list of elements, but take arguments in different orders (that is required for compatibility reasons). The `apply` command also works on matrices (see Section 6.44.6 p.526) and the `map` command also works on polynomials in internal sparse format (see Section 6.27.3 p.351).

- `apply` takes two arguments:
 - f , a function.
 - L , a list.
- `apply(f, L)` returns a list whose elements are $f(x)$ for the elements x of L .

Note that `apply` returns a list (`[]`) even if the second argument is not a list.

Example.

Input:

```
apply(x->sqrt(x), [16,9,4,1])
```

Output:

```
[4,3,2,1]
```

- `map` takes two arguments:
 - L , a list or a polynomial in internal format (see Section 6.27.2 p.351).
 - f , a function.
- `map(L, f)` returns a list whose elements are $f(x)$ for the elements x of L .

Example.

Input:

```
map([16,9,4,1], x->sqrt(x))
```

Output:

```
[4,3,2,1]
```

Examples.

- *Input:*

```
apply(x->x^2, [3,5,1])
```

or:

```
map([3,5,1], x->x^2)
```

(or first define the function $h(x) = x^2$.)

Input:

```
h(x):=x^2
```

then:

```
apply(h, [3,5,1])
```

or:

```
map([3,5,1], h)
```

Output:

```
[9,25,1]
```

- Define the function $g(x) = [x, x^2, x^3]$:
Input:

```
g:=(x)->[x,x^2,x^3]
```

then:

```
apply(g,[3,5,1])
```

or:

```
map([3,5,1],g)
```

Output:

$$\begin{bmatrix} 3 & 9 & 27 \\ 5 & 25 & 125 \\ 1 & 1 & 1 \end{bmatrix}$$

Warning!!!

First purge x if x is not symbolic.

Note that if L_1, L_2 and L_3 are lists, then `sizes([L1, L2, L3])` is equivalent to `map(size, [L1, L2, L3])`.

6.40.29 Applying a bivariate function to the elements of two lists: zip

The `zip` command applies a bivariate function to the elements of 2 lists.

- `zip` takes three arguments:
 - f , a function of two variables.
 - L_1 and L_2 , two lists of the same size n .
- `zip(f, L1, L2)` returns a list of size n whose k th element is f applied to the k th elements of L_1 and L_2

Examples.

- *Input:*

```
zip('sum',[a,b,c,d],[1,2,3,4])
```

Output:

$$[a+1, b+2, c+3, d+4]$$

- *Input:*

```
zip((x,y)->x^2+y^2,[4,2,1],[3,5,1])
```

or:

```
f:=(x,y)->x^2+y^2
zip(f,[4,2,1],[3,5,1])
```

Output:

[25, 29, 2]

- *Input:*

```
f:=(x,y)->[x^2+y^2,x+y]
zip(f,[4,2,1],[3,5,1])
```

Output:

$$\begin{bmatrix} 25 & 7 \\ 29 & 7 \\ 2 & 2 \end{bmatrix}$$

6.40.30 Folding operators: foldl foldr

The **foldl** (left-fold) and **foldr** (right-fold) commands take an infix operator or function of two variables and apply them across a sequence of inputs through left and right association.

- **foldl** takes an arbitrary number of arguments:
 - R , an infix operator or function of two variables.
 - I , an initial value.
 - a_1, a_2, \dots, a_k , an arbitrary number of additional arguments.
- **foldl**(R, I, a_1, \dots, a_k) returns $R(\dots R(R(I, a_1), a_2) \dots, a_k)$.

Example.

Input:

```
foldl('^^',2,3,5)
```

Output:

32768

which is $2^{(3^5)}$

- **foldr** takes an arbitrary number of arguments:
 - R , an infix operator or function of two variables.
 - I , an initial value.
 - a_1, a_2, \dots, a_j , an arbitrary number of additional arguments.
- **foldr**($R, I, a_1, a_2, \dots, a_k$) returns $R(a, \dots (R(a_1, R(a_2, \dots R(a_{k-1}, R(a_k, I))))$

Example.

Input:

```
foldr('^^',2,3,5)
```

Output:

847288609443

which is $3^{(5^2)}$

6.40.31 List of differences of consecutive terms: `deltalist`

The `deltalist` command finds lists of differences.

- `deltalist` takes one argument:
 L , a list.
- `deltalist(L)` returns the list of the difference of all pairs of consecutive terms of L .

Example.

Input:

```
deltalist([5,8,1,9])
```

Output:

 $[3, -7, 8]$
6.40.32 Making a matrix with a list: `list2mat`

The `listmat` command turns a list into a matrix by chopping up the list into separate rows.

- `list2mat` takes two arguments:
 - L , a list.
 - p , a positive integer.
- `list2mat(L, p)` returns the matrix whose first row consists of the first p elements of L , whose next row consists of the next p elements of L , etc. If there are not enough elements to fill up a row, 0s are added.

Examples.

- *Input:*

```
list2mat([5,8,1,9,5,6],2)
```

Output:

$$\begin{pmatrix} 5 & 8 \\ 1 & 9 \\ 5 & 6 \end{pmatrix}$$

- *Input:*

```
list2mat([5,8,1,9],3)
```

Output:

$$\begin{pmatrix} 5 & 8 & 1 \\ 9 & 0 & 0 \end{pmatrix}$$

6.40.33 Making a list with a matrix: `mat2list`

The `mat2list` flattens a matrix into a list. (See also Section 6.40.14 p.474.)

- `mat2list` takes one argument:
 A , a matrix.
- `mat2list(A)` returns the list of the coefficients of A .

Example.

Input:

```
mat2list([[5,8],[1,9]])
```

Output:

```
[5,8,1,9]
```

6.41 Operations on sets and lists**6.41.1 Defining a set or list: `set[] %{} %}`**

Sets and lists are both collections of elements, and so have some operations in common. But lists are different from sets, because for a list, the order is important and the same element can be repeated in a list, while for sets order is not important and each element is unique. See Section 6.40 p.463 for operations on lists.

Recall (see Section 5.3.2 p.98) that to define a set of elements, you can put the elements, separated by commas, within delimiters `%{` and `%}` or `set[` and `]`.

Example.

Input:

```
set[1,2,3,4]
```

or:

```
%{1,2,3,4%}
```

Output:

```
[[1,2,3,4]]
```

Also, (see Section 5.3.3 p.98) to define a list of elements, you can put the elements, separated by commas, within delimiters `[` and `]`. Lists are also called vectors.

Example.

Input:

```
[1,2,5]
```

Output:

```
[1,2,5]
```

6.41.2 Testing if a value is in a list or a set: member contains

The `member` and `contains` commands determine whether or not an object is in a list; the difference between them is the order of the arguments (required for compatibility reasons).

- `member` takes two arguments:
 - c , a value.
 - L , a list or a set.
- `member(c, L)` returns 0 if c is not an element of L and otherwise returns n if c is in L and its first position has index $n - 1$. 0 if c is not in L .

Examples.

- *Input:*

```
member(2, [0, 1, 2, 3, 4, 2])
```

Output:

3

- *Input:*

```
member(2, %{0, 1, 2, 3, 4, 2%})
```

Output:

3

- `contains` takes two arguments:
 - L , a list or a set.
 - c , a value.
- `contains(L, c)` returns 0 if c is not an element of L and otherwise returns n if c is in L and its first position has index $n - 1$. 0 if c is not in L .

Examples.

- *Input:*

```
contains([0, 1, 2, 3, 4, 2], 2)
```

Output:

3

- *Input:*

```
contains(%{0, 1, 2, 3, 4, 2%}, 2)
```

Output:

3

6.41.3 Union of two sets or of two lists: union

The `union` operator is an infix operator that finds the union of the elements of two sets or lists; the result will always be a set.

Examples.

- *Input:*

```
set[1,2,3,4] union set[5,6,3,4]
```

or:

```
%{1,2,3,4%} union %{5,6,3,4%}
```

Output:

```
[[1,2,3,4,5,6]]
```

- *Input:*

```
[1,2,3] union [2,5,6]
```

Output:

```
[[1,2,3,5,6]]
```

6.41.4 Intersection of two sets or of two lists: intersect

The `intersect` operator is an infix operator that can find the intersection of the elements of two sets or lists; the result will always be a set.

Examples.

- *Input:*

```
set[1,2,3,4] intersect set[5,6,3,4]
```

or:

```
%{1,2,3,4%} intersect %{5,6,3,4%}
```

Output:

```
[[3,4]]
```

- *Input:*

```
[1,2,3,4] intersect [5,6,3,4]
```

Output:

```
[[3,4]]
```

6.41.5 Difference of two sets or of two lists: `minus`

The `minus` operator is an infix operator that can find the set difference of the elements of two sets or lists; the result will always be a set.

Examples.

- *Input:*

```
set[1,2,3,4] minus set[5,6,3,4]
```

or:

```
%{1,2,3,4%} minus %{5,6,3,4%}
```

Output:

```
[[1,2]]
```

- *Input:*

```
[1,2,3,4] minus [5,6,3,4]
```

Output:

```
[[1,2]]
```

Cartesian products

Defining an n -tuple: `tuple`

To define an n -tuple (rather than a list of n objects), you can put the elements, separated by commas, inside the delimiters `tuple[` and `]`.

Example.

Input:

```
set[tuple[1,3,4],tuple[1,3,5],tuple[2,3,4]]
```

Output:

```
[[tuple[1,3,4],tuple[1,3,5],tuple[2,3,4]]]
```

The Cartesian product of two sets: `*`

You can compute the Cartesian product of two sets with the infix `*` operator.

Examples.

- *Input:*

```
set[1,2] * set[3,4]
```

Output:

```
[[tuple[1,3],tuple[1,4],tuple[2,3],tuple[2,4]]]
```

- *Input:*

```
set[1,2] * set[3,4] * set[5,6]
```

Output:

```
[[tuple[1,3,5],tuple[1,3,6],tuple[1,4,5],tuple[1,4,6],tuple[2,3,5],tuple[2,3,6],tuple[2,4,5],tuple[2,4,6]]]
```

6.42 Functions for vectors

6.42.1 Norms of a vector: `maxnorm` `l1norm` `l2norm` `norm`

There are different norms for vectors in \mathbb{R}^n , and **Xcas** has different commands for them.

Given a list $L = [a_1, \dots, a_n]$,

- `maxnorm(L)` returns the l^∞ norm of L , namely $\max(|a_1|, |a_2|, \dots, |a_n|)$

Example.

Input:

```
maxnorm([3,-4,2])
```

Output:

4

Indeed, $4 = \max(|3|, |-4|, |2|)$.

- `l1norm(L)` returns the l^1 norm of L , namely $|a_1| + |a_2| + \dots + |a_n|$ coordinates.

Example.

Input:

```
l1norm([3,-4,2])
```

Output:

9

Indeed, $9 = |3| + |-4| + |2|$.

- `norm(L)` or `l2norm(L)` returns the ℓ^2 norm of L ; namely $\sqrt{|a_1|^2 + |a_2|^2 + \dots + |a_n|^2}$.

Example.

Input:

```
norm([3,-4,2])
```

Output:

$\sqrt{29}$

Indeed, $29 = |3|^2 + |-4|^2 + |2|^2$.

6.42.2 Normalizing a vector: `normalize` `unitV`

The `normalize` command finds the unit vector in the direction of a given vector.

`unitV` is a synonym for `normalize`.

- `normalize` takes one argument:
 V , a vector (list).

- `normalize(V)` normalizes this vector for the l^2 norm (the square root of the sum of the squares of its coordinates).

Example.*Input:*`normalize([3,4,5])`*Output:*

$$\left[\frac{3}{5\sqrt{2}}, \frac{4}{5\sqrt{2}}, \frac{5}{5\sqrt{2}} \right]$$

6.42.3 Term by term sum of two lists: + .+

The infix operators `+` and `.+` as well as the prefixed operator `'+'` return the term by term sum of two lists. If the two lists do not have the same size, the smaller list is padded with zeros.

Note the difference with sequences: if the infix operator `+` or the prefixed operator `'+'` is applied to two sequences, it merges the sequences, hence return the sum of all the terms of the two sequences.

Examples.

- *Input:*

`[1,2,3]+[4,3,5]`*or:*`[1,2,3] .+[4,3,5]`*or:*`'+'([1,2,3],[4,3,5])`*Output:*`[5,5,8]`

- *Input:*

`[1,2,3,4,5,6]+[4,3,5]`*or:*`[1,2,3,4,5,6] .+[4,3,5]`*or:*`'+'([1,2,3,4,5,6],[4,3,5])`*Output:*`[5,5,8,4,5,6]`**Warning!**

When the operator `+` is prefixed, it should be quoted (`'+'`).

6.42.4 Term by term difference of two lists: - . -

The infix operators - and . - as well as the prefixed operator '- ' return the term by term difference of two lists. If the two lists do not have the same size, the smaller list is padded with zeros.

Example.

Input:

$$[1, 2, 3] - [4, 3, 5]$$

or:

$$[1, 2, 3] \text{ . - } [4, 3, 5]$$

or:

$$'-'([1, 2, 3], [4, 3, 5])$$

or:

$$'-'([[1, 2, 3], [4, 3, 5]])$$

Output:

$$[-3, -1, -2]$$

Warning!

When the operator - is prefixed, it should be quoted ('-').

6.42.5 Term by term product of two lists: .*

The infix operator .* returns the term by term product of two lists of the same size.

Example.

Input:

$$[1, 2, 3] \text{ .* } [4, 3, 5]$$

Output:

$$[4, 6, 15]$$
6.42.6 Term by term quotient of two lists: ./

The infix operator ./ returns the term by term quotient of two lists of the same size.

Example.

Input:

$$[1, 2, 3] \text{ ./ } [4, 3, 5]$$

Output:

$$\left[\frac{1}{4}, \frac{2}{3}, \frac{3}{5} \right]$$

6.42.7 Scalar product : `scalar_product` * `dotprod` `dot` `dotP` `scalar_Product`

The `dot` command finds the dot product of two vectors.

`dotP`, `dotprod`, `scalar_product`, and `scalarProduct` are synonyms for `dot`.

The infix operator `*` and its prefixed form `'*'` will also find dot products.

- `dot` takes two arguments:
 V_1 and V_2 , two lists (vectors) of the same size.
- `dot(V_1, V_2)` returns the dot product of V_1 and V_2 .

Example.

Input:

```
dot([1,2,3],[4,3,5])
```

or:

```
scalar_product([1,2,3],[4,3,5])
```

or:

```
[1,2,3]*[4,3,5]
```

or:

```
'*'( [1,2,3],[4,3,5] )
```

Output:

```
25
```

Indeed $25 = 1 \cdot 4 + 2 \cdot 3 + 3 \cdot 5$.

Note that `*` may be used to find the product of two polynomials represented as list of their coefficients, but to avoid ambiguity, the polynomial lists must be `poly1[...]`.

6.42.8 Cross product: `cross` `crossP` `crossproduct`

The `cross` command finds the cross product of two vectors.

`crossP` and `crossproduct` are synonyms for `cross`.

- `cross` takes two arguments:
 V_1 and V_2 , two vectors of length 3.
- `cross(V_1, V_2)` returns the cross product of V_1 and V_2 .

Example.

Input:

```
cross([1,2,3],[4,3,2])
```

Output:

```
[-5,10,-5]
```

Indeed, $-5 = 2 \cdot 2 - 3 \cdot 3$, $10 = -1 \cdot 2 + 4 \cdot 3$, $-5 = 1 \cdot 3 - 2 \cdot 4$.

6.42.9 Statistics on lists: mean variance stddev stddevp median quantile quartiles boxwhisker

The functions described here can be used if a statistics series is contained in a list. See also Section 6.45.16 p.535 for statistics on matrices and Section 9 p.711 for more general statistics.

Let L be a list.

- `mean(L)` computes the arithmetic mean of a list.

Examples.

– *Input:*

```
mean([3,4,2])
```

Output:

3

– *Input:*

```
mean([1,0,1])
```

Output:

$\frac{2}{3}$

- `stddev(L)` computes the standard deviation of a population, for the population L .

Example.

Input:

```
stddev([3,4,2])
```

Output:

$\frac{\sqrt{6}}{3}$

- `stddevp(L)` computes an unbiased estimate of the standard deviation of the population for the sample L . The following relation holds:

$$\text{stddevp}(L)^2 = \frac{\text{size}(L) \cdot \text{stddev}(L)^2}{\text{size}(L) - 1}$$

Example.

Input:

```
stddevp([3,4,2])
```

Output:

1

- `variance(L)` computes the variance of L , which is the square of `stddevp(L)`.

Example.

Input:

```
variance([3,4,2])
```

Output:

$$\frac{2}{3}$$

- `median(L)` computes the median of L .

Example.

Input:

```
median([0,1,3,4,2,5,6])
```

Output:

3.0

- `quantile(L, d)` computes the deciles of L , where d is the decile.

Examples.

– *Input:*

```
quantile([0,1,3,4,2,5,6],0.25)
```

Output (the first quartile):

1.0

– *Input:*

```
quantile([0,1,3,4,2,5,6],0.5)
```

Output (the median):

3.0

– *Input:*

```
quantile([0,1,3,4,2,5,6],0.75)
```

Output (the third quartile):

5.0

- `quartiles(L)` returns a list consisting of the minimum, the first quartile, the median, the third quartile and the maximum of L .

Example.

Input:

```
quartiles([0,1,3,4,2,5,6])
```

Output:

$$\begin{bmatrix} 0.0 \\ 1.0 \\ 3.0 \\ 5.0 \\ 6.0 \end{bmatrix}$$

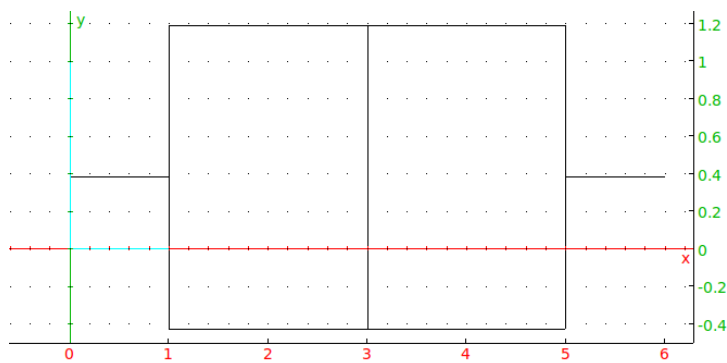
- `boxwhisker(L)` draws the whisker box of a statistics series stored in L .

Example.

Input:

```
boxwhisker([0,1,3,4,2,5,6])
```

Output:



Example.

Define the list **A** by:

Input:

```
A:=[0,1,2,3,4,5,6,7,8,9,10,11] ;;
```

Then:

Input:

```
mean(A)
```

Output:

$$\frac{11}{2}$$

Input:

```
stddev(A)
```

Output:

$$\frac{2}{12}\sqrt{429}$$

Input:

```
quantile(A,0.1)
```

Output:

1.0

Input:

`quantile(A,0.25)`

Output:

2.0

Input:

`median(A)`

or:

`quantile(A,0.5)`

Output:

5.0

Input:

`quantile(A,0.75)`

Output:

8.0

Input:

`quantile(A,0.9)`

Output:

10.0

Input:

`max(A)`

Output:

11

Input:

`quartiles(A)`

Output:

$$\begin{bmatrix} 0.0 \\ 2.0 \\ 5.0 \\ 8.0 \\ 11.0 \end{bmatrix}$$

6.43 Tables with strings as indices: `table`

A table is a map (associative container) used to store information associated to indices which are much more general than integers, such as strings or sequences. For example, you can use one to store a table of phone numbers indexed by names.

In `Xcas`, the indices in a table may be any kind of `Xcas` objects. Access is done by a binary search algorithm, where the sorting function first sorts by `type` then uses an order for each type (e.g. `<` for numeric types, lexicographic order for strings, etc.)

The `table` command creates a table.

- `table` takes an unspecified number of arguments:
seq, a list or sequence of equalities of the form *index_name=element_value*.
- `table(seq)` returns a table. The elements of the table can be retrieved using index bracket notation; If *T* is the name of the table, then `T(index_name)` returns *element_value*.

Example.

Input:

```
T:=table(3=-10,"a"=10,"b"=20,"c"=30,"d"=40);;
```

Input:

```
T["b"]
```

Output:

```
20
```

Input:

```
T[3]
```

Output:

```
-10
```

Remark.

Tables can be created and the elements of a table can be changed using the `:=` assignment.

- If *T* is a symbolic variable then the assignment `T(index_name:=element_value)` will create a table *T* with one element.
- If *n* is an integer, then the assignment `T(n):=obj` will do the following:
 - If the variable *T* was assigned to a list or a sequence, then the *n*th element of *T* is modified.
 - if the variable *T* was not assigned, a table *T* is created with one entry (corresponding to the index *n*). Note that after the assignment *T* is not a list, despite the fact that *n* is an integer.

6.44 Matrices

6.44.1 Matrices

A matrix is represented by a list of lists, all having the same size.

Example.

Input:

```
[[1,2,3],[4,5,6]]
```

Output:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

You can give a matrix a name with assignment.

Input:

```
A:= [[1,2,6],[3,4,8],[1,0,1]]
```

Output:

$$\begin{bmatrix} 1 & 2 & 6 \\ 3 & 4 & 8 \\ 1 & 0 & 1 \end{bmatrix}$$

6.44.2 Special matrices

Identity matrix: `idn` identity

The `idn` command finds identity matrices.

- `idn` takes one argument:
 n , a positive integer or
 A , a square matrix.
- `idn(n)` returns the $n \times n$ identity matrix.
- `idn(A)` returns the identity matrix the same size as A .

Examples.

- *Input:*

```
idn(3)
```

Output:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

- *Input:*

```
idn([[2,3],[4,5]])
```

Output:

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Zero matrix: newMat

The `newMat` command creates a matrix of all 0s.

- `newMat` takes two arguments:
 n and p , two positive integers. `newMat(n, p)` returns the $n \times p$ zero matrix.

Example.

Input:

```
newMat(4,4)
```

Output:

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Diagonals of matrices and diagonal matrices: BlockDiagonal diag

The `diag` command either creates a diagonal matrix or finds the diagonal elements of an existing matrix.

`BlockDiagonal` is a synonym for `diag`.

- `diag` takes one argument:
 L , a list or a square matrix.
- `diag(L)` (for a list L) returns the diagonal matrix with the entries of L on the diagonal.
- `diag(L)` (for a matrix L) returns a list consisting of the diagonal elements of L .

Examples.

- *Input:*

```
diag([1,4])
```

Output:

$$\begin{bmatrix} 1 & 0 \\ 0 & 4 \end{bmatrix}$$

- *Input:*

```
diag([[1,2],[3,4]])
```

Output:

$$[1, 4]$$

Jordan block: `JordanBlock`

The `JordanBlock` command creates a Jordan Block; i.e., a square matrix with the same value for all diagonal elements, 1s just above the diagonal, and 0s everywhere else.

- `JordanBlock` takes two arguments:

- a , an expression.
- n , a positive integer.

`JordanBlock(a, n)` returns the $n \times n$ matrix with a s on the principal diagonal, 1s above this diagonal and 0s everywhere else.

Example.

Input:

`JordanBlock(7,3)`

Output:

$$\begin{bmatrix} 7 & 1 & 0 \\ 0 & 7 & 1 \\ 0 & 0 & 7 \end{bmatrix}$$

Hilbert matrix: `hilbert`

A Hilbert matrix is a square matrix whose element in the i th row and j th column (recall the numbering starting at 0) is

$$a_{j,k} = \frac{1}{j+k+1}$$

The `hilbert` command finds Hilbert matrices.

- `hilbert` takes one argument:
 n , a positive integer.
- `hilbert(n)` returns the $n \times n$ Hilbert matrix.

Example.

Input:

`hilbert(4)`

Output:

$$\begin{pmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \end{pmatrix}$$

Vandermonde matrix: `vandermonde`

A Vandermonde matrix is a square matrix where each row starts with a 1 and is in geometric progression. The `vandermonde` command finds a Vandermonde matrix.

- `vandermonde` takes one argument:
 $X = [x_0, \dots, x_{n-1}]$, a vector.
- `vandermonde(X)` returns the corresponding Vandermonde matrix; namely, the k -th row of the matrix is the vector whose components are x_i^k for $i = 0..n - 1$ and $k = 0..n - 1$.

Warning!

The indices of the rows and columns begin at 0 with `Xcas`.

Example.

Input:

```
vandermonde([a,2,3])
```

Output (if a is symbolic else `purge(a)`):

$$\begin{pmatrix} 1 & a & aa \\ 1 & 2 & 4 \\ 1 & 3 & 9 \end{pmatrix}$$

6.44.3 Combining matrices**Making a matrix with a list of matrices: `blockmatrix`**

The `blockmatrix` combines several matrices into one larger matrix.

- `blockmatrix` takes three arguments:
 - m and n , two positive integers.
 - L , a list of $m \cdot n$ matrices such that the first m matrices have the same number of rows; the next m matrices have the same number of rows, etc; and the number of columns in each group of m matrices is the same (for example, all the matrices in L could have the same dimension), so that the n groups of m matrices can be stacked above each other to form a larger matrix.
 - `blockmatrix(m,n,L)` returns the larger matrix formed by the matrices in L by putting each group of m matrices next to each other, and stacking the resulting n matrices on top of each other.
- If the matrices in L each have the same dimension $p \times q$, the result will be a matrix with dimension $p * n \times q * m$.

Examples.

- *Input:*

```
blockmatrix(2,3,[idn(2),idn(2),idn(2),
idn(2),idn(2),idn(2)])
```

Output:

$$\begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}$$

• *Input:*

```
blockmatrix(3,2,[idn(2),idn(2),
idn(2),idn(2),idn(2),idn(2)])
```

Output:

$$\begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

• *Input:*

```
blockmatrix(2,2,[idn(2),newMat(2,3), newMat(3,2),idn(3)])
```

Output:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

• *Input:*

```
blockmatrix(3,2,[idn(1),newMat(1,4),
newMat(2,3),idn(2),newMat(1,2),[[1,1,1]]])
```

Output:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

Input:

```
A:=[[1,1],[1,1]];B:=[[1],[1]];;
blockmatrix(2,3,[2*A,3*A,4*A,5*B,newMat(2,4),6*B])
```

Output:

$$\begin{bmatrix} 2 & 2 & 3 & 3 & 4 & 4 \\ 2 & 2 & 3 & 3 & 4 & 4 \\ 5 & 0 & 0 & 0 & 0 & 6 \\ 5 & 0 & 0 & 0 & 0 & 6 \end{bmatrix}$$

Making a matrix from two matrices: `semi_augment`

The `semi_augment` command concatenates two matrices with the same number of columns.

- `semi_augment` takes two arguments:
 A and B , two matrices with the same number of columns.
- `semi_augment(A, B)` returns the matrix which has the rows of A followed by the rows of B .

Examples.

- *Input:*

```
semi_augment([[3,4],[2,1],[0,1]],[[1,2],[4,5]])
```

Output:

$$\begin{bmatrix} 3 & 4 \\ 2 & 1 \\ 0 & 1 \\ 1 & 2 \\ 4 & 5 \end{bmatrix}$$

```
[[3,4],[2,1],[0,1],[1,2],[4,5]]
```

- *Input:*

```
semi_augment([[3,4,2]],[[1,2,4]])
```

Output:

$$\begin{bmatrix} 3 & 4 & 2 \\ 1 & 2 & 4 \end{bmatrix}$$

Note the difference with `concat`.

Input:

```
concat([[3,4,2]],[[1,2,4]])
```

Output:

$$[3 \ 4 \ 2 \ 1 \ 2 \ 4]$$

Indeed, when the two matrices A and B have the same dimension, `concat` makes a matrix with the same number of rows as A and B by gluing them side by side.

Input:

```
concat([[3,4],[2,1],[0,1]],[[1,2],[4,5]])
```

Output:

$$\begin{bmatrix} 3 & 4 \\ 2 & 1 \\ 0 & 1 \\ 1 & 2 \\ 4 & 5 \end{bmatrix}$$

but input:

`concat([[3,4],[2,1]], [[1,2],[4,5]])`

Output:

$$\begin{bmatrix} 3 & 4 & 1 & 2 \\ 2 & 1 & 4 & 5 \end{bmatrix}$$

Making a matrix from two matrices: `augment` `concat`

The `augment` command glues two matrices, either side by side or one on top of the other.

Here, `concat` can be used as a synonym for `augment`.

- `augment` has two arguments:
 A and B , two matrices with the same number of rows or the same number of columns.
- `augment(A, B)` returns the matrix consisting of:
 - if A and B have the same number of rows, then the matrix being returned consists of the columns of A followed by the columns of B ; in other words, A and B are glued side by side.
 - if A and B do not have the same number of rows but have the same number of columns, then the matrix being returned consists of the rows of A followed by the rows of B ; in other words, A and B are glued one on top of the other.

Examples.

- *Input:*

`augment([[3,4,5],[2,1,0]], [[1,2],[4,5]])`

Output:

$$\begin{bmatrix} 3 & 4 & 5 & 1 & 2 \\ 2 & 1 & 0 & 4 & 5 \end{bmatrix}$$

- *Input:*

`augment([[3,4],[2,1],[0,1]], [[1,2],[4,5]])`

Output:

$$\begin{bmatrix} 3 & 4 \\ 2 & 1 \\ 0 & 1 \\ 1 & 2 \\ 4 & 5 \end{bmatrix}$$

• *Input:*

```
augment([[3,4,2]], [[1,2,4]])
```

Output:

$$\begin{bmatrix} 3 & 4 & 2 & 1 & 2 & 4 \end{bmatrix}$$

Note that if A and B have the same dimension, then `augment(A, B)` will return a matrix with the same number of rows as A and B by horizontal gluing. In that case, if you want to combine them by vertical gluing, you must use `semi_augment(A, B)`.

Appending a column to a matrix: `border`

The `border` command adds a column to a matrix.

- `border` takes two arguments:
 - A , a matrix.
 - b , a list whose length equals the number of rows of A .
- `border(A, L)` returns a matrix equal to A with the transpose of L forming an additional column to the right; so

```
border(A,b)=tran([op(tran(A)),b])=tran(append(tran(A),b))
```

Examples.

• *Input:*

```
border([[1,2,4],[3,4,5]], [6,7])
```

Output:

$$\begin{bmatrix} 1 & 2 & 4 & 6 \\ 3 & 4 & 5 & 7 \end{bmatrix}$$

• *Input:*

```
border([[1,2,3,4],[4,5,6,8],[7,8,9,10]], [1,3,5])
```

Output:

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 1 \\ 4 & 5 & 6 & 8 & 3 \\ 7 & 8 & 9 & 10 & 5 \end{bmatrix}$$

6.44.4 Creating a matrix with a formula or function: `makemat` `matrix`

You can use a function or a formula to specify the elements of a matrix with the `makemat` or `matrix` command.

- `makemat` takes three arguments:
 - f , a function of two variables j and k which returns the value of $a_{j,k}$, the element at row index j and column index k of the resulting matrix.
 - n and p , two positive integers.
- `makemat`(f, n, p) returns the $n \times p$ matrix $A = (a_{j,k})$ with $a_{j,k} = f(j, k)$ for $j = 1..n$ and $k = 1..p$.

Example.

Input:

```
makemat((j,k)->j+k,4,3)
```

or:

```
h(j,k):=j+k
makemat(h,4,3)
```

Output:

$$\begin{pmatrix} 0 & 1 & 2 \\ 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5 \end{pmatrix}$$

Note that the indices are counted starting from 0.

The `matrix` command can be used similarly, but note that the arguments are given in a different order and the indices start at 1.

(`matrix` can also be used to turn tables into matrices; see Section 6.46.1 p.539.)

- `matrix` takes two mandatory arguments and one optional argument:
 - n and p , two integers.
 - Optionally, f , a function of two variables j and k which should return the value of $a_{j,k}$, the element at row index j and column index k of the resulting matrix.
- `matrix`(n, p) returns the $n \times p$ matrix consisting of all 0s.
- `matrix`(n, p, f) returns the $n \times p$ matrix $A = (a_{j,k})$ with $a_{j,k} = f(j, k)$ for $j = 1..n$ and $k = 1..p$.

Examples.

- *Input:*

```
matrix(2,3)
```

Output:

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

• *Input:*

```
matrix(4,3,(j,k)->j+k)
```

or:

```
h(j,k):=j+k
matrix(4,3,h)
```

Output:

$$\begin{pmatrix} 0 & 1 & 2 \\ 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5 \end{pmatrix}$$

6.44.5 Getting the parts of a matrix

Accessing parts of a matrix: `[]` at

The rows of a matrix are the elements of a list, and can be accessed with indices using the postfix `[...]` or the prefix `at` (see Section 6.39.6 p.459).

Example.

Input:

```
A:= [[1,2,6], [3,4,8], [1,0,1]]
```

then:

```
A[0]
```

or:

```
at(A,0)
```

Output:

```
[1,2,6]
```

To extract a column of a matrix, you can first turn the columns into rows with `transpose` (see Section 6.47.1 p.541), then extract the row as above.

Example.

Input:

```
tran(A)[1]
```

or:

```
at(tran(A),1)
```

Output:

```
[2,4,0]
```

Individual elements are simply elements of the rows.

Example.

Input:

```
A[0][1]
```

Output:

```
2
```

This can be abbreviated by listing the row and column separated by a comma.

Input:

```
A[0,1]
```

or:

```
at(A,[0,1])
```

Output:

```
2
```

The indexing begins with 0; you can have the indices start with 1 by enclosing them in double brackets.

Input:

```
A[[1,2]]
```

Output:

```
2
```

You can use a range (see Section 6.37.1 p.445) of indices to get submatrices.

Examples.

- *Input:*

```
A[1,0..2]
```

Output:

```
[3,4,8]
```

- *Input:*

```
A[0..2,1]
```

Output:

```
[2,4,0]
```


- *Input:*

$$A[0..2, 1..2]$$

Output:

$$\begin{bmatrix} 2 & 6 \\ 4 & 8 \\ 0 & 1 \end{bmatrix}$$

- *Input:*

$$A[0..1, 1..2]$$

Output:

$$\begin{bmatrix} 2 & 6 \\ 4 & 8 \end{bmatrix}$$

- This gives you another way to extract a full column, by specifying all the rows as an index interval.

$$A[0..2, 1]$$

Output:

$$[2, 4, 0]$$

Recall that An index of -1 returns the last element of a list, an index of -2 the second to last element, etc.

Examples.

- *Input:*

$$A[-1]$$

Output:

$$[1, 0, 1]$$

- *Input:*

$$A[1, -1]$$

Output:

Extracting rows or columns of a matrix (Maple compatibility): `row`
`col`

The `row` (respectively `col`) command extracts one or several rows (respectively columns) of a matrix.

- `row` takes two arguments:
 - A , a matrix.
 - r , a row index or a range $n_1..n_2$.
- `row(A, r)` returns the row or sequence of rows given by r .

Examples.

- *Input:*

$$\text{row}([[1, 2, 3], [4, 5, 6], [7, 8, 9]], 1)$$

Output:

$$[4, 5, 6]$$

- *Input:*

$$\text{row}([[1, 2, 3], [4, 5, 6], [7, 8, 9]], 0..1)$$

Output:

$$[1, 2, 3], [4, 5, 6]$$

- `col` takes two arguments:
 - A , a matrix.
 - c , a column index or a range $n_1..n_2$.
- `col(A, c)` returns the column or sequence of columns given by c .

Examples.

- *Input:*

$$\text{col}([[1, 2, 3], [4, 5, 6], [7, 8, 9]], 1)$$

Output:

$$[2, 5, 8]$$

- *Input:*

$$\text{col}([[1, 2, 3], [4, 5, 6], [7, 8, 9]], 0..1)$$

Output:

$$[1, 4, 7], [2, 5, 8]$$

Extracting a sub-matrix of a matrix (TI compatibility): subMat

The `subMat` command finds submatrices of a matrix.

- `subMat` takes one mandatory argument and four optional arguments:
 - A , a matrix.
 - Optionally, r_1 , an integer, the row index for the beginning of the submatrix (by default, $r_1 = 0$).
 - Optionally, c_1 , an integer, the column index for the beginning of the submatrix (by default, $c_1 = 0$).
 - Optionally, r_2 , an integer, the row index for the end of the submatrix (by default, r_2 equals one less than the number of rows of A).
 - Optionally, c_2 , an integer, the column index for the end of the submatrix (by default, c_2 equals one less than the number of columns of A).
- `subMat(A , $\langle r_1, c_1, r_2, c_2 \rangle$)` returns the sub-matrix of A from position (r_1, c_1) to (r_2, c_2) .

Example.

Input:

$$A := [[3, 4, 5], [1, 2, 6]]$$

Output:

$$\begin{bmatrix} 3 & 4 & 5 \\ 1 & 2 & 6 \end{bmatrix}$$

- *Input:*

$$\text{subMat}(A, 0, 1, 1, 2)$$

Output:

$$\begin{bmatrix} 4 & 5 \\ 2 & 6 \end{bmatrix}$$

- *Input:*

$$\text{subMat}(A, 0, 1, 1, 1)$$

Output:

$$\begin{bmatrix} 4 \\ 2 \end{bmatrix}$$

$$[[4], [2]]$$

- *Input:*

$$\text{subMat}(A, 1)$$

or:

```
subMat(A,1,0)
```

or:

```
subMat(A,1,0,1)
```

or:

```
subMat(A,1,0,1,2)
```

Output:

```
[ 1  2  6 ]
```

6.44.6 Modifying matrices

Modifying matrix elements by assignment: :=

You can change the elements of a named matrix by assignment (see Section 5.4.2 p.100).

Example.

Input:

```
A:= [[1,2,6], [3,4,8], [1,0,1]]
```

then:

```
A[0,1] := 5;;  
A
```

Output:

```
[ 1  5  6 ]  
[ 3  4  8 ]  
[ 1  0  1 ]
```

Recall that the elements are indexed starting at 0, using double brackets allows you to use indices starting at 1.

Input:

```
A[[1,2]]:=7;;  
A
```

Output:

```
[ 1  7  6 ]  
[ 3  4  8 ]  
[ 1  0  1 ]
```

You can use assignment to change several entries of a matrix at one.

Example.

Create a diagonal matrix with a diagonal of [1,2,3]:

Input:

```
M:= matrix(3,3)
```

Output:

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

Input:

```
M[0..2,0..2] := [1,2,3]
```

Output:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{pmatrix}$$

To make the last column [4,5,6]:

Input:

```
M[0..2,2] := [4,5,6]
```

Output:

$$\begin{pmatrix} 1 & 0 & 4 \\ 0 & 2 & 5 \\ 0 & 0 & 6 \end{pmatrix}$$

Modifying matrix elements by reference: := =<

When you change an element of a matrix with the := assignment, a new copy of the matrix is created with the modified element. Particularly for large matrices, it is more efficient to use the =< assignment (see Section 5.4.3 p.101), which will change the element of the matrix without making a copy.

Example.

Input:

```
A:= [[4,5],[2,6]]
```

The following commands will all return the matrix A with the element in the second row, first column, changed to 3.

Input:

```
A[1,0] := 3
```

or:

```
A[1,0] =< 3
```

or:

```
A[[2,1]] := 3
```

or:

```
A[[2,1]] =< 3
```

then:

A

Output:

$$\begin{bmatrix} 4 & 5 \\ 3 & 6 \end{bmatrix}$$

You can change larger parts of a matrix simultaneously.

Example.

Input:

A:= [[4,5],[2,6]]

The following commands will change the second row to [3,7]

Input:

A[1]:= [3,7]

or:

A[1] =< [3,7]

or:

A[[2]]:= [3,7]

or:

A[[2]] =< [3,7]

Output:

$$\begin{bmatrix} 4 & 5 \\ 3 & 7 \end{bmatrix}$$

The =< assignment must be used carefully, since it not only modifies a matrix A, it modifies all objects pointing to A. In a program, initialization should contain a line like A:= copy(B) (see Section 5.4.4 p.101) so modifications done on A don't affect B, and modifications done on B don't affect A.

For example:

Input:

B:= [[4,5],[2,6]]

then:

A:= B

or:

A =< B

creates two matrices equal to

$$\begin{bmatrix} 4 & 5 \\ 2 & 6 \end{bmatrix}$$

Input:

```
A[1] =< [3,7]
```

or:

```
B[1] =< [3,7]
```

transforms both A and B to

$$\begin{bmatrix} 4 & 5 \\ 3 & 7 \end{bmatrix}$$

On the other hand, creating A and B with:

Input:

```
B:= [[4,5],[2,6]]
A:= copy(B)
```

will again create two matrices equal to

$$\begin{bmatrix} 4 & 5 \\ 2 & 6 \end{bmatrix}$$

But:

Input:

```
A[1] =< [3,7]
```

will change A to

$$\begin{bmatrix} 4 & 5 \\ 3 & 7 \end{bmatrix}$$

but B will still be

$$\begin{bmatrix} 4 & 5 \\ 2 & 6 \end{bmatrix}$$

Modifying an element or a row of a matrix: subsop

The **subsop** command modifies elements of lists (see Section 6.40.7 p.468), and so you can use it to modify elements or rows of matrices. It is used mainly for Maple and MuPAD compatibility, and the argument list is in a different order in Maple mode. Unlike := or =<, it does not require the matrix to be stored in a variable.

Let A be the matrix give by:

Input:

```
A:= [[4,5],[2,6]]
```

In Xcas, Mupad and TI modes: Recall that the indexing in Xcas mode begins with 0, while in Mupad and TI modes it begins with 1.

To modify an element:

- **subsop** takes two arguments:
 - A, a matrix.

- $[r, c]=v$, an equality between a matrix position (given as a list) and a value.

The two sides of the equality can also be given as separate arguments.

`subsop(A, [r, c]=v)` returns the matrix which is the same as A except that the element in row r , column c is now v .

Examples.

- *Input (in Xcas mode):*

```
subsop([[4,5],[2,6]], [1,0]=3)
```

or:

```
subsop([[4,5],[2,6]], [1,0], 3)
```

Output:

$$\begin{bmatrix} 4 & 5 \\ 3 & 6 \end{bmatrix}$$

- *Input (in Mupad or TI mode):*

```
subsop([[4,5],[2,6]], [2,1]=3)
```

or:

```
subsop([[4,5],[2,6]], [2,1], 3)
```

Output:

$$\begin{bmatrix} 4 & 5 \\ 3 & 6 \end{bmatrix}$$

To modify a row:

- `subsop` takes two arguments:

- A , a matrix.
- $r = L$, an equality between a row index and a list with the same length as the rows of A .

The two sides of the equality can also be given as separate arguments.

- `subsop(A, r = L)` returns the matrix which is the same as A except that row r is now equal to the list L .

Examples.

- *Input (in Xcas mode):*


```
subsop([[4,5],[2,6]],1=[3,3])
```

or:

```
subsop([[4,5],[2,6]],1,[3,3])
```

Output:

$$\begin{bmatrix} 4 & 5 \\ 3 & 3 \end{bmatrix}$$

- *Input (in Mupad or TI mode):*

```
subsop([[4,5],[2,6]],2=[3,3])
```

or:

```
subsop([[4,5],[2,6]],2,[3,3])
```

Output:

$$\begin{bmatrix} 4 & 5 \\ 3 & 3 \end{bmatrix}$$

In Maple mode: Recall that the indexing in Maple mode begins with 1.
To modify an element:

- **subsop** takes two arguments:
 - $[r,c]=v$, an equality between a matrix position (given as a list) and a value.
The two sides of the equality can also be given as separate arguments.
 - A , a matrix.

subsop $([r,c]=v,A)$ returns the matrix which is the same as A except that the element in row r , column c is now v .

Example.

Input:

```
subsop([2,1]=3,[[4,5],[2,6]])
```

Output:

$$\begin{bmatrix} 4 & 5 \\ 3 & 6 \end{bmatrix}$$

To modify a row:

- **subsop** takes two arguments:
 - $r = L$, an equality between a row index and a list with the same length as the rows of the second argument A .
 - A , a matrix.

- **subsop**($r = L, A$) returns the matrix which is the same as A except that row r is now equal to the list L .

Example:

Input (in Maple mode):

```
subsop(2=[3,3],[[4,5],[2,6]])
```

Output:

$$\begin{bmatrix} 4 & 5 \\ 3 & 3 \end{bmatrix}$$

In all modes: If the matrix is stored in a variable, for example with the matrix A as above, it is easier to enter $A[1,0]:=3$ and $A[1]=[3,3]$ to modify A as above.

Also, note that **subsop** with a 'n=NULL' argument deletes row number n .

Example.

Input (in Xcas mode):

```
subsop([[4,5],[2,6]],'1=NULL')
```

Output:

$$\begin{bmatrix} 4 & 5 \end{bmatrix}$$

Removing rows or columns of a matrix: delrows delcols

The **delrows** (respectively **delcols**) command removes one or more rows (respectively columns) from a matrix.

- **delrows** takes two arguments:
 - A , a matrix.
 - r , an integer or a range of integers.
- **delrows**(A, r) returns the matrix equal to A with the row(s) given by r removed.

Examples.

- *Input:*

```
delrows([[1,2,3],[4,5,6],[7,8,9]],1)
```

Output:

$$\begin{bmatrix} 1 & 2 & 3 \\ 7 & 8 & 9 \end{bmatrix}$$

- *Input:*

```
delrows([[1,2,3],[4,5,6],[7,8,9]],0..1)
```

Output:

$$\begin{bmatrix} 7 & 8 & 9 \end{bmatrix}$$

The `delcols` command behaves like `delrows`, but for columns.

- `delcols` takes two arguments:
 - A , a matrix.
 - c , an integer or a range of integers.
- `delrows(A, c)` returns the matrix equal to A with the column(s) given by c removed.

Examples.

- *Input:*

```
delcols([[1,2,3],[4,5,6],[7,8,9]],1)
```

Output:

$$\begin{bmatrix} 1 & 3 \\ 4 & 6 \\ 7 & 9 \end{bmatrix}$$

- *Input:*

```
delcols([[1,2,3],[4,5,6],[7,8,9]],0..1)
```

Output:

$$\begin{bmatrix} 3 \\ 6 \\ 9 \end{bmatrix}$$

Resizing a matrix or vector: `REDIM redim`

The `REDIM` command resizes matrices and vectors.

`redim` is a synonym for `REDIM`.

For matrices:

- `REDIM` takes two arguments:
 - A , a matrix.
 - $[m, n]$, a list of two positive integers.
- `REDIM($A, [m, n]$)` returns A resized to an $m \times n$ matrix, removing elements (if necessary) to make it smaller and adding 0s (if necessary) to make it larger.

Examples.

- *Input:*

`REDIM([[4, 1, -2], [1, 2, -1], [2, 1, 0]], [5, 4])`

Output:

$$\begin{pmatrix} 4 & 1 & -2 & 0 \\ 1 & 2 & -1 & 0 \\ 2 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

• *Input:*

`REDIM([[4, 1, -2], [1, 2, -1], [2, 1, 0]], [2, 1])`

Output:

$$\begin{pmatrix} 4 \\ 1 \end{pmatrix}$$

For vectors:

• `REDIM` takes two arguments:

- L , a list.
- n , a positive integer.

• `REDIM(L, n)` returns L resized to a list of length n , removing elements (if necessary) to make it smaller and adding 0s (if necessary) to make it larger.

Examples.

• *Input:*

`REDIM([4, 1, -2, 1, 2, -1], 10)`

Output:

`[4, 1, -2, 1, 2, -1, 0, 0, 0, 0]`

• *Input:*

`REDIM([4, 1, -2, 1, 2, -1], 3)`

Output:

`[4, 1, -2]`

Replacing part of a matrix or vector: REPLACE replace

The REPLACE command replaces part of a matrix or vector.

replace is a synonym for **REPLACE**.

For matrices:

- REPLACE takes three arguments:
 - A , a matrix.
 - $[m, n]$, a list of two positive integers.
 - B , a matrix.

$\text{REPLACE}(A, [m, n], B)$ returns the matrix equal to A but with the upper left corner of B placed at row m , column n , replacing the previous elements of A . The matrix B will be shrunk, if necessary, to fit.

Examples.

- *Input:*

```
REPLACE([[1,2,3],[4,5,6]],[0,1],[[5,6],[7,8]])
```

Output:

$$\begin{pmatrix} 1 & 5 & 6 \\ 4 & 7 & 8 \end{pmatrix}$$

- *Input:*

```
REPLACE([[1,2,3],[4,5,6]],[1,2],[[7,8],[9,0]])
```

Output:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 7 \end{pmatrix}$$

For lists:

- REPLACE takes three arguments:
 - L , a list.
 - n , a positive integer.
 - M , another list.

$\text{REPLACE}(L, n, M)$ returns the list equal to L but with the elements beginning at index n replaced by the elements of M , replacing the previous elements of L . The list M will be shrunk, if necessary, to fit.

Examples.

- *Input:*

```
REPLACE([4,1,-2,1,2,-1],2,[10,11])
```

Output:

$[4, 1, 10, 11, 2, -1]$

- *Input:*

`REPLACE([4, 1, -2, 1, 2, -1], 1, [10, 11, 13])`

Output:

$[4, 10, 11, 13, 2, -1]$

Applying a function to the elements of a matrix: `apply`

The `apply` command can apply a function to the elements of a matrix. (See Section 6.40.28 p.484 for other uses of `apply`.)

- `apply` takes three arguments:
 - f , a function of one variable.
 - A , a matrix.
 - `matrix`, the symbol.

`apply(f, A, matrix)` returns a matrix whose elements are $f(x)$ for the elements x of A .

Example.

Input:

`apply(x->x^2, [[1, 2, 3], [4, 5, 6]], matrix)`

Output:

$$\begin{bmatrix} 1 & 4 & 9 \\ 16 & 25 & 36 \end{bmatrix}$$

6.45 Arithmetic and matrices

6.45.1 Evaluating a matrix: `evalm`

The `evalm` command is used in Maple to evaluate a matrix. In Xcas, matrices are evaluated by default, the command `evalm` is only available for compatibility, it is equivalent to `eval` (see Section 6.12.1 p.203).

6.45.2 Addition and subtraction of two matrices: `+` `-` `.*` `.-`

The infix operators `+` and `.*` (resp. `-` and `.-`) are used for the addition (resp. subtraction) of two matrices.

Examples.

- *Input:*

`[[1, 2], [3, 4]] + [[5, 6], [7, 8]]`

Output:

$$\begin{bmatrix} 6 & 8 \\ 10 & 12 \end{bmatrix}$$

Input:

$$[[1, 2], [3, 4]] - [[5, 6], [7, 8]]$$

Output:

$$\begin{bmatrix} -4 & -4 \\ -4 & -4 \end{bmatrix}$$

Remark.

+ and - can be used as prefixed operators; in this case they must be quoted, '+' and '-' (see Section 6.42.3 p.494 and Section 6.42.4 p.495).

Examples.

- *Input:*

$$'+ ' ([[1, 2], [3, 4]], [[5, 6], [7, 8]], [[2, 2], [3, 3]])$$

Output:

$$\begin{bmatrix} 8 & 10 \\ 13 & 15 \end{bmatrix}$$

- *Input:*

$$'-' ([[1, 2], [3, 4]], [[5, 6], [7, 8]])$$

Output:

$$\begin{bmatrix} -4 & -4 \\ -4 & -4 \end{bmatrix}$$

6.45.3 Multiplication of two matrices: * &*

The infix operator * and &* are used for the multiplication of two matrices.

Example.

Input:

$$[[1, 2], [3, 4]] * [[5, 6], [7, 8]]$$

or:

$$[[1, 2], [3, 4]] \&* [[5, 6], [7, 8]]$$

Output:

$$\begin{bmatrix} 19 & 22 \\ 43 & 50 \end{bmatrix}$$

6.45.4 Addition of elements of a column of a matrix: `sum`

The `sum` command (see also Section 6.20.3 p.282) can add the elements of the columns of a matrix.

- `sum` takes one argument:
 A , a matrix.
- `sum(A)` returns the list whose elements are the sum of the elements of each column of the matrix A .

Example.

Input:

$$\text{sum}([[1, 2], [3, 4]])$$

Output:

$$[4, 6]$$

6.45.5 Cumulated sum of elements of each column of a matrix: `cumSum`

The `cumSum` command finds the cumulated sum of each column of a matrix (see also Section 6.40.26 p.481).

- `cumSum` takes one argument:
 A , a matrix.
- `cumSum(A)` returns the matrix whose columns are the cumulated sum of the elements of the corresponding column of the matrix A .

Example.

Input:

$$\text{cumSum}([[1, 2], [3, 4], [5, 6]])$$

Output:

$$\begin{bmatrix} 1 & 2 \\ 4 & 6 \\ 9 & 12 \end{bmatrix}$$

since the cumulated sums of the first column are: 1, 1+3=4, 1+3+5=9 and the accumulated sums of the second column are: 2, 2+4=6, 2+4+6=12.

6.45.6 Multiplication of elements of each column of a matrix: `product`

The `product` command can multiply the elements of the columns of a matrix (see Section 6.40.27 p.482 for other things `product` can do).

- `product` takes one argument:
 A , a matrix.

- `product(A)` returns the list whose elements are the product of the elements of each column of the matrix A .

Example.

Input:

```
product([[1,2],[3,4]])
```

Output:

```
[3,8]
```

6.45.7 Power of a matrix: `^` & `^`

The infix operator `^` (or `&^`) is used to raise a matrix to an integral power.

Example.

Input:

```
[[1,2],[3,4]] ^ 5
```

or:

```
[[1,2],[3,4]] &^ 5
```

Output:

```
[ 1069  1558 ]
[ 2337  3406 ]
```

6.45.8 Hadamard product: `hadamard product .*`

The `hadamard` command can find the Hadamard product of two matrices; namely, the term-by-term product of the two matrices.

The `product` command can do the same thing (see also Section 6.40.27 p.482 for other uses of `product`).

- `hadamard` takes two arguments:
 A and B , two matrices of the same size.
- `hadamard(A,B)` returns the matrix where each element is the product of the corresponding elements of A and B .

The infix operator `.*` also finds the Hadamard product, and also works on lists.

Examples.

- *Input:*

```
hadamard([1, 2],[3,4]],[[5, 6],[7, 8]])
```

or:

```
hadamard([1, 2],[3,4]],[[5, 6],[7, 8]])
```

or:

$$[[1, 2], [3, 4]] \text{ .* } [[5, 6], [7, 8]]$$

Output:

$$\begin{pmatrix} 5 & 12 \\ 21 & 32 \end{pmatrix}$$

• *Input:*

$$[1, 2, 3, 4] \text{ .* } [5, 6, 7, 8]$$

Output:

$$[5, 12, 21, 32]$$

6.45.9 Hadamard division: ./

The infix operator ./ finds the Hadamard quotient of two matrices or lists A and B of the same size; namely, it returns the matrix or the list where each element is the term by term quotient of the corresponding elements of A and B .

Example.

Input:

$$[[1, 2], [3, 4]] \text{ ./ } [[5, 6], [7, 8]]$$

Output:

$$\begin{bmatrix} \frac{1}{5} & \frac{1}{3} \\ \frac{3}{7} & \frac{1}{2} \end{bmatrix}$$

6.45.10 Hadamard power: .^

The infix operator ./ finds the Hadamard power of a matrix or list A to a real number b ; namely, it returns the matrix or the list where each element is the corresponding element of A raised to the b th power.

Example.

Input:

$$[[1, 2], [3, 4]] \text{ .}^2$$

Output:

$$\begin{bmatrix} 1 & 4 \\ 9 & 16 \end{bmatrix}$$

6.45.11 The elementary row operations

Adding a row to another row: rowAdd

The `rowAdd` command adds one row of a matrix to another row.

• `rowAdd` takes three arguments:

- A , a matrix.
- n_1 and n_2 , two integers.

- `rowAdd(A, n_1, n_2)` returns the matrix obtained by replacing in A , the row of index n_2 by the sum of the rows of index n_1 and n_2 .

Example.

Input:

```
rowAdd([[1,2],[3,4]],0,1)
```

Output:

$$\begin{bmatrix} 1 & 2 \\ 4 & 6 \end{bmatrix}$$

Multiplying a row by an expression: `mRow` `scale` `SCALE`

The `mRow`, `scale` and `SCALE` commands multiply a row of a matrix by an expression.

- `mRow` takes three arguments:
 - $expr$, an expression.
 - A , a matrix.
 - n , an integer.
- `mRow($expr, A, n$)` returns the matrix obtained by replacing in A , the row of index n by the product of the row of index n by $expr$.

Example.

Input:

```
mRow(12,[[1,2],[3,4]],1)
```

Output:

$$\begin{bmatrix} 1 & 2 \\ 36 & 48 \end{bmatrix}$$

The `scale` command is the same as `mRow` except that it takes the arguments in a different order.

`SCALE` is a synonym for `scale`.

- `scale` takes three arguments:
 - A , a matrix.
 - $expr$, an expression.
 - n , an integer.
- `scale($A, expr, n$)` returns the matrix obtained by replacing in A , the row of index n by the product of the row of index n by $expr$.

Example.

Input:

```
scale([[1,2],[3,4]],12,1)
```

Output:

$$\begin{bmatrix} 1 & 2 \\ 36 & 48 \end{bmatrix}$$

Adding k times a row to an another row: mRowAdd scaleadd SCALEADD

The `mRowAdd`, `scaleadd` and `SCALEADD` commands add a multiple of one row of a matrix to another.

- `mRowAdd` takes four arguments:
 - k , a real number.
 - A , a matrix.
 - n_1 and n_2 , two integers.
- `mRowAdd(k, A, n_1, n_2)` returns the matrix obtained by replacing in A , the row with index n_2 by the sum of the row with index n_2 and k times the row with index n_1 .

Example.

Input:

```
mRowAdd(1.1, [[5,7],[3,4],[1,2]],1,2)
```

Output:

$$\begin{bmatrix} 5 & 7 \\ 3 & 4 \\ 4.3 & 6.4 \end{bmatrix}$$

The `scaleadd` command is the same as `mRowAdd` except that it takes the arguments in a different order.

`SCALEADD` is a synonym for `scaleadd`.

- `scaleadd` takes four arguments:
 - A , a matrix.
 - k , a real number.
 - n_1 and n_2 , two integers.
- `scaleadd(A, k, n_1, n_2)` returns the matrix obtained by replacing in A , the row with index n_2 by the sum of the row with index n_2 and k times the row with index n_1 .

Example.

Input:

```
scaleadd([[5,7],[3,4],[1,2]],1.1,1,2)
```

Output:

$$\begin{bmatrix} 5 & 7 \\ 3 & 4 \\ 4.3 & 6.4 \end{bmatrix}$$

Exchanging two rows: rowSwap rowswap swaprow

The `rowSwap` command switches two rows in a matrix. `rowswap` and `swaprow` are synonyms for `rowSwap`.

- `rowSwap` takes three arguments:
 - A , a matrix.
 - n_1 and n_2 , integers.
- `rowSwap(A, n_1, n_2)` returns the matrix obtained by exchanging in A , the row with index n_1 with the row with index n_2 .

Example.

Input:

```
rowSwap([[1,2],[3,4]],0,1)
```

Output:

$$\begin{bmatrix} 3 & 4 \\ 1 & 2 \end{bmatrix}$$

Exchanging two columns: colSwap colswap swapcol

The `colSwap` command switches two columns in a matrix. `colswap` and `swapcol` are synonyms for `colSwap`.

- `colSwap` takes three arguments:
 - A , a matrix.
 - n_1 and n_2 , integers.
- `colSwap(A, n_1, n_2)` returns the matrix obtained by exchanging in A , the column with index n_1 with the column with index n_2 .

Example.

Input:

```
colSwap([[1,2],[3,4]],0,1)
```

Output:

$$\begin{bmatrix} 2 & 1 \\ 4 & 3 \end{bmatrix}$$

6.45.12 Counting the elements of a matrix satisfying a property: count

The `count` applies a function to the elements of a matrix or list and adds the result. Hence, if the function is a boolean function, then `count` will count the number of elements satisfying the property that the function tests for.

- `count` takes two arguments:
 - f , a real-valued function.

- A , a matrix or a list.
- `count(f, A)` returns the sum of the function f applied to the elements of A .

Examples.

- *Input:*

```
count(x->x, [[2,12], [45,3], [7,78]])
```

Output:

147

Indeed: $2+12+45+3+7+78=147$.

- *Input:*

```
count(x->x<10, [[2,12], [45,3], [7,78]])
```

Output:

3

6.45.13 Counting the elements equal to a given value: count_eq

The `count_eq` command counts the number of elements in a matrix equal to a given value.

- `count_eq` takes two arguments:
 - a , a value.
 - A , a matrix or a list.
- `count_eq(a, A)` returns the number of elements of A that are equal to a .

Example.

Input:

```
count_eq(12, [[2,12,45], [3,7,78]])
```

Output:

6.45.14 Counting the elements smaller than a given value:
count_inf

The `count_inf` command counts the number of elements in a matrix less than a given value.

- `count_inf` takes two arguments:
 - a , a real number.
 - A , a matrix or a list of real numbers.
- `count_inf(a, A)` returns the number of elements of A that are strictly less than a .

Example.

Input:

```
count_inf(12, [2, 12, 45, 3, 7, 78])
```

Output:

3

6.45.15 Counting the elements greater than a given value:
count_sup

The `count_sup` command counts the number of elements in a matrix greater than a given value.

- `count_sup` takes two arguments:
 - a , a real number.
 - A , a matrix or a list of real numbers.
- `count_sup(a, A)` returns the number of elements of A that are strictly greater than a .

Example.

Input:

```
count_sup(12, [[2, 12, 45], [3, 7, 78]])
```

Output:

2

**6.45.16 Statistics functions acting on column matrices: mean
stddev variance median quantile quartiles boxwhisker**

The following functions can find statistics for the columns of a matrix. See also Section 6.42.9 p.497 for statistics on lists and Section 9 p.711 for more general statistics.

Let A be a matrix.

- **mean**(A) computes the arithmetic means of the columns of the matrix A .

Examples.

– *Input:*

$$\text{mean}([[3, 4, 2], [1, 2, 6]])$$

Output:

$$[2, 3, 4]$$

– *Input:*

$$\text{mean}([[1, 0, 0], [0, 1, 0], [0, 0, 1]])$$

Output:

$$\left[\frac{1}{3}, \frac{1}{3}, \frac{1}{3} \right]$$

- **stddev**(A) computes the standard deviations for the populations given by the columns of A .

Example.

Input:

$$\text{stddev}([[3, 4, 2], [1, 2, 6]])$$

Output:

$$[1, 1, 2]$$

- **stddevp**(A) computes the unbiased estimates of the standard deviations of the populations for the samples given by the columns of A .

Example.

Input:

$$\text{stddevp}([[3, 4, 2], [1, 2, 6]])$$

Output:

$$[\sqrt{2}, \sqrt{2}, 2\sqrt{2}]$$

- **variance**(A) computes the variances of the columns of A .

Example.

Input:

$$\text{variance}([[3, 4, 2], [1, 2, 6]])$$

Output:

$$[1, 1, 4]$$

- **median**(A) computes the medians of the columns of A .

Example.

Input:


```
median([[6,0,1,3,4,2,5],[0,1,3,4,2,5,6],[1,3,4,2,5,6,0],
       [3,4,2,5,6,0,1],[4,2,5,6,0,1,3],[2,5,6,0,1,3,4]])
```

Output:

```
[2.0, 2.0, 3.0, 3.0, 2.0, 2.0, 3.0]
```

- `quantile(A, d)` computes the deciles of the columns of A , where d is the decile.

Examples.

– *Input:*

```
quantile([[6,0,1,3,4,2,5],[0,1,3,4,2,5,6],[1,3,4,2,5,6,0],
         [3,4,2,5,6,0,1],[4,2,5,6,0,1,3],[2,5,6,0,1,3,4]], 0.25)
```

Output (the first quartiles of the columns):

```
[1.0, 1.0, 2.0, 2.0, 1.0, 1.0, 1.0]
```

– *Input:*

```
quantile([[6,0,1,3,4,2,5],[0,1,3,4,2,5,6],[1,3,4,2,5,6,0],
         [3,4,2,5,6,0,1],[4,2,5,6,0,1,3],[2,5,6,0,1,3,4]], 0.75)
```

Output (the third quartiles of the columns):

```
[4.0, 4.0, 5.0, 5.0, 5.0, 5.0, 5.0]
```

- `quartiles(A)` returns a matrix where each column consists of the minimum, the first quartile, the median, the third quartile and the maximum of the corresponding column of A .

Example.

Input:

```
quartiles([[6,0,1,3,4,2,5],[0,1,3,4,2,5,6],[1,3,4,2,5,6,0],
          [3,4,2,5,6,0,1],[4,2,5,6,0,1,3],[2,5,6,0,1,3,4]])
```

Output:

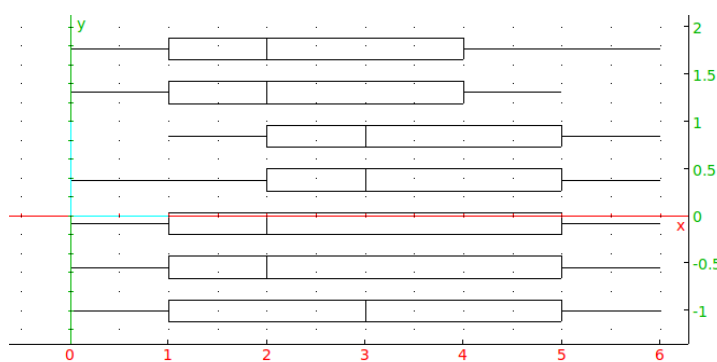
$$\begin{bmatrix} 0.0 & 0.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 1.0 & 1.0 & 2.0 & 2.0 & 1.0 & 1.0 & 1.0 \\ 2.0 & 2.0 & 3.0 & 3.0 & 2.0 & 2.0 & 3.0 \\ 4.0 & 4.0 & 5.0 & 5.0 & 5.0 & 5.0 & 5.0 \\ 6.0 & 5.0 & 6.0 & 6.0 & 6.0 & 6.0 & 6.0 \end{bmatrix}$$

The output is a matrix, its first row is the minima of each column, its second row is the first quartiles of each column, its third row the medians of each column, its fourth row the third quartiles of each column and its last row the maxima of each column:

- `boxwhisker(L)` draws the whisker box of the statistics series stored in the columns of A .

Example.*Input:*

```
boxwhisker([[6,0,1,3,4,2,5],[0,1,3,4,2,5,6],
            [1,3,4,2,5,6,0],[3,4,2,5,6,0,1],
            [4,2,5,6,0,1,3],[2,5,6,0,1,3,4]])
```

Output:**6.45.17 Dimension of a matrix: dim**

The `dim` command finds the dimension of a matrix.

- `dim` takes one argument:
 A , a matrix
- `dim(A)` returns a list of the number of rows and columns of the matrix A .

Example.*Input:*

```
dim([[1,2,3],[3,4,5]])
```

Output:

```
[2,3]
```

6.45.18 Number of rows: rowdim rowDim nrows

The `rowdim` command finds the number of rows of a matrix.
`rowDim` and `nrows` are synonyms for `rowdim`.

- `rowdim` takes one argument:
 A , a matrix.
- `rowdim(A)` returns the number of rows of the matrix A .

Example.*Input:*

```
rowdim([[1,2,3],[3,4,5]])
```

Output:

2

6.45.19 Number of columns: coldim colDim ncols

The `coldim` command finds the number of columns of a matrix. `colDim` and `ncols` are synonyms for `coldim`.

- `coldim` takes one argument:
 A , a matrix.
- `coldim(A)` returns the number of columns of the matrix A .

Example.*Input:*

```
coldim([[1,2,3],[3,4,5]])
```

Output:

3

6.46 Sparse matrices**6.46.1 Defining sparse matrices**

A matrix is *sparse* if most of its elements are 0. To specify a sparse matrix, it is easier to define the non-zero elements. This can be done with a table (see Section 6.43 p.501). The `matrix` command (see Section 6.44.4 p.510) or the `convert` command (see Section 6.23.26 p.323) can then turn the table into a matrix.

Example.

First, define the non-zero elements.

Input:

```
A:= table((0,0)=1, (1,1)=2, (2,2)=3, (3,3) = 4, (4,4) = 5)
```

or:

```
purge(A)
A[0..4,0..4]:=[1,2,3,4,5]
```

Output:

Key	Value
(0,0)	1
(1,1)	2
(2,2)	3
(3,3)	4
(4,4)	5

This table can be converted to a matrix with either the `convert` command or the `matrix` command.

Input:

```
a:= convert(A,array)
```

or:

```
a:= matrix(A)
```

Output:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 & 5 \end{bmatrix}$$

6.46.2 Operations on sparse matrices

All matrix operations can be done on tables that are used to define sparse matrices.

Example.

Create some sparse matrices.

Input:

```
purge(A) ;;
A[0..2,0..2] := [1,2,3]
```

Output:

Key	Value
(0,0)	1
(1,1)	2
(2,2)	3

Input:

```
purge(B) ;;
B[0..1,1..2] := [1,2] ;;
B[0..2,0] := 5
```

Output:

Key	Value
(0,0)	5
(0,1)	1
(1,0)	5
(1,2)	2
(2,0)	5

The usual operations will work on A and B.

Input:

```
A + B
```

Output:

Key	Value
(0, 0)	6
(0, 1)	1
(1, 0)	5
(1, 1)	2
(1, 2)	2
(2, 0)	5
(2, 2)	3

Input:

`A * B`

Output:

Key	Value
(0, 0)	5
(0, 1)	1
(1, 0)	10
(1, 2)	4
(2, 0)	15

Input:

`2*A`

Output:

Key	Value
(0, 0)	2
(1, 1)	4
(2, 2)	6

6.47 Linear algebra

6.47.1 Transpose of a matrix: `tran` transpose

The `tran` command finds the transpose of a matrix.
`transpose` is a synonym for `tran`.

- `tran` takes one argument:
 A , a matrix.
- `tran(A)` returns the transpose matrix of A .

Example.

Input:

`tran([[1,2],[3,4]])`

Output:

$$\begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}$$

6.47.2 Inverse of a matrix: inv /

The `inv` command finds the inverse of a matrix.

- `inv` takes one argument:
 A , a matrix.
- `inv(A)` returns the inverse matrix of A .

Note that $1/A$ is another way to find the inverse of a matrix.

Example.

Input:

$$\text{inv}(\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix})$$

or:

$$1/\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

or:

$$A:=\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}; 1/A$$

Output:

$$\begin{bmatrix} -2 & 1 \\ \frac{3}{2} & -\frac{1}{2} \end{bmatrix}$$

6.47.3 Trace of a matrix: trace

The *trace* of a square matrix is the sum of the diagonal elements. The `trace` command finds the trace of a matrix.

- `trace` takes one argument:
 A , a matrix.
- `trace(A)` returns the trace of the matrix A .

Example.

Input:

$$\text{trace}(\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix})$$

Output:

5

6.47.4 Determinant of a matrix: det

The `det` command finds the determinant of a matrix.

- `det` takes one mandatory argument and one optional argument:
 - A , a matrix.
 - Optionally, *method*, which determines how the determinant will be computed and can be one of:

- * **lagrange** When the matrix elements are polynomials or rational functions, this method computes the determinant by evaluating the elements and using Lagrange interpolation.
- * **rational_det** This method uses Gaussian elimination without converting to the internal format for fractions.
- * **bareiss** This uses the Gauss-Bareiss algorithm.
- * **linsolve** This uses the p -adic algorithm for matrices with integer coefficients.
- * **minor_det** This uses expansion by minor determinants. This requires 2^n operations, but can still be faster for average sized matrices (up to about $n = 20$).

- `det(A, method)` returns the determinant of the matrix A .

Examples.

- *Input:*

```
det([[1,2],[3,4]])
```

Output:

-2

- *Input:*

```
det(idn(3))
```

Output:

1

6.47.5 Determinant of a sparse matrix: `det_minor`

The `det_minor` command finds the determinant of a matrix by expanding the determinant using Laplace's algorithm.

- `det_minor` takes one argument:
 A , a matrix.
- `det_minor(A)` returns the determinant of the matrix A .

Examples.

- *Input:*

```
det_minor([[1,2],[3,4]])
```

Output:

-2

- *Input:*

```
det_minor(idn(3))
```

Output:

1

6.47.6 Rank of a matrix: rank

The **rank** command finds the rank of a matrix.

- **rank** takes one argument:
 A , a matrix.
- **rank**(A) returns the rank of the matrix A .

Examples.

- *Input:*

$$\text{rank}(\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix})$$

Output:

$$2$$

- *Input:*

$$\text{rank}(\begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix})$$

Output:

$$1$$
6.47.7 Transconjugate of a matrix: trn

The *transconjugate* of a matrix is the conjugate of the transpose of the matrix. The **trn** command finds the transconjugate of a matrix.

- **trn** takes one argument:
 A , a matrix.
- **trn**(A) returns the transconjugate of A .

Example.

Input:

$$\text{trn}(\begin{bmatrix} i & 1+i \\ 1 & 1-i \end{bmatrix})$$

Output:

$$\begin{pmatrix} -i & 1 \\ 1-i & 1+i \end{pmatrix}$$

6.47.8 Equivalent matrix: changebase

The **changebase** command changes a matrix to represent the same linear function in a different basis.

- **changebase** takes two arguments:
 - A , a matrix.
 - P , a change-of-basis matrix.

- `changebase(A, P)` returns the matrix $P^{-1}AP$.

Examples.

- *Input:*

```
changebase([[1, 2], [3, 4]], [[1, 0], [0, 1]])
```

Output:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

- *Input:*

```
changebase([[1, 1], [0, 1]], [[1, 2], [3, 4]])
```

Output:

$$\begin{bmatrix} -5 & -8 \\ \frac{9}{2} & 7 \end{bmatrix}$$

Indeed:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}^{-1} \cdot \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} -5 & -8 \\ \frac{9}{2} & 7 \end{bmatrix}$$

6.47.9 Basis of a linear subspace : `basis`

The `basis` command finds a basis of a linear subspace of \mathbb{R}^n given a spanning set.

- `basis` takes one argument:
 L , a list of vectors generating a linear subspace of \mathbb{R}^n .
- `basis(L)` returns a list of vectors that is a basis of this linear subspace.

Example.

Input:

```
basis([[1, 2, 3], [1, 1, 1], [2, 3, 4]])
```

Output:

$$\{[-1, 0, 1], [0, -1, -2]\}$$

6.47.10 Basis of the intersection of two subspaces: `ibasis`

The `ibasis` command finds a basis for the intersection of two subspaces of \mathbb{R}^n .

- `ibasis` takes two arguments:
 L_1 and L_2 , two lists of vectors generating two subspaces of \mathbb{R}^n .
- `ibasis(L1, L2)` returns a list of vectors forming a basis for the intersection of these two subspaces.

Example.*Input:*

```
ibasis([[1,2]], [[2,4]])
```

Output:

$$\{[1,2]\}$$
6.47.11 Image of a linear function: image

The **image** command finds a basis for the image of a linear function.

- **image** takes one argument:
 A , a matrix representing a linear function with respect to the standard basis.
- **image**(A) returns a list of vectors that is a basis of the image of the linear function.

Example.*Input:*

```
image([[1,1,2],[2,1,3],[3,1,4]])
```

Output:

$$\begin{bmatrix} -1 & 0 & 1 \\ 0 & -1 & -2 \end{bmatrix}$$
6.47.12 Kernel of a linear function: kernel nullspace ker

The **ker** command finds a basis for the kernel of a linear function.

kernel and **nullspace** are synonyms for **ker**.

- **ker** takes one argument:
 A , a matrix representing a linear function with respect to the standard basis.
- **ker**(A) returns a list of vectors that is a basis of the kernel of the linear function.

Example.*Input:*

```
ker([[1,1,2],[2,1,3],[3,1,4]])
```

Output:

$$\begin{bmatrix} 1 & 1 & -1 \end{bmatrix}$$

6.47.13 Kernel of a linear function: Nullspace

The `Nullspace` command is the inert form of `nullspace`. **Warning:** The `Nullspace` command is only useful in Maple mode. (See Section 3.5.2 p.71; you can get into Maple mode by hitting the state line red button then `Prog` style, then choosing Maple and Apply).

- `Nullspace` takes one argument:
 A , an integer matrix representing a linear function with respect to the standard basis.
- `Nullspace(A) mod p` returns a list of vectors that is a basis for the kernel of the linear transformation $\mathbb{Z}/p\mathbb{Z}[X]$.

Examples.

- *Input:*

```
Nullspace([[1,1,2],[2,1,3],[3,1,4]])
```

Output:

$$\text{nullspace} \left(\begin{bmatrix} 1 & 1 & 2 \\ 2 & 1 & 3 \\ 3 & 1 & 4 \end{bmatrix} \right)$$

- *Input (in Maple mode):*

```
Nullspace([[1,2],[3,1]]) mod 5
```

Output:

$$[2, -1]$$

In Xcas mode, the equivalent input is:

Input (in Xcas mode):

```
nullspace([[1,2],[3,1]] % 5)
```

Output:

$$\begin{bmatrix} 2 \% 5 & -1 \end{bmatrix}$$

6.47.14 Subspace generated by the columns of a matrix: colspace

The `colspace` command finds a basis for the column space of a matrix.

- `colspace` takes one mandatory argument and one optional argument:
 - A , a matrix.
 - Optionally, *var*, a variable name.

- `colspace(A $\langle var \rangle$)` returns a matrix whose columns are a basis of the subspace generated by the columns of A . With the optional argument var , `Xcas` will store the dimension of the subspace generated by the columns of A .

Examples.

- *Input:*

```
colspace([[1,1,2],[2,1,3],[3,1,4]])
```

Output:

$$\begin{bmatrix} -1 & 0 \\ 0 & -1 \\ 1 & -2 \end{bmatrix}$$

- *Input:*

```
colspace([[1,1,2],[2,1,3],[3,1,4]],dimension)
```

Output:

$$\begin{bmatrix} -1 & 0 \\ 0 & -1 \\ 1 & -2 \end{bmatrix}$$

then input:

```
dimension
```

Output:

2

6.47.15 Subspace generated by the rows of a matrix: rowSPACE

The `rowSPACE` command finds a basis for the row space of a matrix.

- `rowSPACE` takes one mandatory argument and one optional argument:
 - A , a matrix.
 - Optionally, var , a variable name.
- `rowSPACE(A $\langle var \rangle$)` returns a list of vectors which form a basis of the subspace generated by the rows of A . With the optional argument var , `Xcas` will store the dimension of the subspace generated by the rows of A .

Examples.

- *Input:*

```
rowSPACE([[1,1,2],[2,1,3],[3,1,4]])
```

Output:

$$\begin{bmatrix} -1 & 0 & -1 \\ 0 & -1 & -1 \end{bmatrix}$$

• *Input:*

```
rowSpace([[1,1,2],[2,1,3],[3,1,4]],dimension)
```

Output:

$$\begin{bmatrix} -1 & 0 & -1 \\ 0 & -1 & -1 \end{bmatrix}$$

then input:

```
dimension
```

Output:

2

6.48 Matrix reduction

6.48.1 Eigenvalues: eigenvals

The `eigenvals` command finds eigenvalues of a matrix.

- `eigenvals` takes one argument:
 A , a square matrix.
- `eigenvals(A)` returns the sequence of the eigenvalues of A , including multiplicities. (If A is an $n \times n$ matrix, then the sequence will have n values.)

Remark: Xcas may not be able to find the exact roots of the characteristic polynomial in some cases. In that case, `eigenvals(A)` will return approximate eigenvalues of A if the coefficients are numeric or a subset of the eigenvalues if the coefficients are symbolic.

Examples.

• *Input:*

```
eigenvals([[4,1,-2],[1,2,-1],[2,1,0]])
```

Output:

2, 2, 2

• *Input:*

```
eigenvals([[4,1,0],[1,2,-1],[2,1,0]])
```

Output:

$$\frac{\text{rootof}([1, 0, -20, 0, 100], [1, 0, -24, 0, 144, 0, -148])}{18}, \frac{\text{rootof}([-1, 0, 20, 18, 8], [1, 0, -24, 0, 144, 0, -148])}{36}$$

Input:

```
evalf(eigenvals([[4, 1, 0], [1, 2, -1], [2, 1, 0]]))
```

Output:

```
1.46081112719, 4.21431974338, 0.324869129433
```

6.48.2 Eigenvalues: `egvl` eigenvalues `eigVl`

The `egvl` command finds the Jordan form of a matrix. `eigenvalues` and `eigVl` are synonyms for `egvl`.

- `egvl` takes one argument:
 A , a square matrix.
- `egvl(A)` returns the Jordan normal form of A .

Examples.

- *Input:*

```
egvl([[4, 1, -2], [1, 2, -1], [2, 1, 0]])
```

Output:

$$\begin{bmatrix} 2 & 1 & 0 \\ 0 & 2 & 1 \\ 0 & 0 & 2 \end{bmatrix}$$

- *Input:*

```
egvl([[4, 1, 0], [1, 2, -1], [2, 1, 0]])
```

Output:

See Section 6.27.21 p.365 for a discussion of `rootof`.

$$\begin{bmatrix} \frac{\text{rootof}([1, 0, -20, 0, 100], [1, 0, -24, 0, 144, 0, -148])}{18} & 0 & \text{rootof}([-1, 0, 20, 18, 8], [1, 0, -24, 0, 144, 0, -148])}{36} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Input:

```
evalf(egvl([[4, 1, 0], [1, 2, -1], [2, 1, 0]]))
```

Output:

$$\begin{bmatrix} 1.46081112719 & 0.0 & 0.0 \\ 0.0 & 4.21431974338 & 0.0 \\ 0.0 & 0.0 & 0.324869129433 \end{bmatrix}$$

6.48.3 Eigenvectors: `egv` `eigenvectors` `eigenvecs` `eigVc`

The `egv` command finds the eigenvectors of a diagonalizable matrix. `eigenvectors`, `eigenvecs` and `eigVc` are synonyms for `egv`.

- `egv` takes one argument:
 A , a square matrix.
- `egv(A)` returns a matrix whose columns are the eigenvectors of the matrix A if A is diagonalizable, otherwise it will fail.

See also Section 6.48.5 p.553 for characteristic vectors.

Examples.

- *Input:*

```
egv([[1,1,3],[1,3,1],[3,1,1]])
```

Output:

$$\begin{bmatrix} 1 & -1 & 1 \\ 1 & 2 & 0 \\ 1 & -1 & -1 \end{bmatrix}$$

- *Input:*

```
egv([[4,1,-2],[1,2,-1],[2,1,0]])
```

Output:

```
"Not diagonalizable at eigenvalue 2"
```

- *Input (in complex mode):*

```
egv([[2,0,0],[0,2,-1],[2,1,2]])
```

Output:

$$\begin{bmatrix} 1 & 0 & 0 \\ -2 & -1 & -1 \\ 0 & -i & i \end{bmatrix}$$

6.48.4 Rational Jordan matrix: `rat_jordan`

The `rat_jordan` command finds the rational Jordan form of a matrix.

- `rat_jordan` takes one mandatory and one optional argument:
 - A , a square matrix (preferably with exact coefficients).
 - Optionally, *var*, a variable name.

- `rat_jordan(A)` (in all modes but **Maple**) returns a sequence $[P, J]$ of two matrices, where J is the rational Jordan matrix of A (the most reduced matrix in the field of the coefficients of A or the complexified field in complex mode) and

$$J = P^{-1}AP$$

The coefficients of P and J belongs to the same field as the coefficients of A . If A is diagonalizable in the field of its coefficients, then the columns of P are the eigenvectors of A .

`rat_jordan(A)` (in **Maple** mode) only returns the matrix J .

- `rat_jordan(A⟨var⟩)` returns the matrix J , as above, and assigns the matrix P to the variable var .

Examples.

- *Input (not in Maple mode):*

```
rat_jordan([[1,0,0],[1,2,-1],[0,0,1]])
```

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}, \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- *Input:*

```
rat_jordan([[1,0,0],[1,2,-1],[0,0,1]],P)
```

$$\begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

then input:

`P`

Output:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

- *Input:*

```
rat_jordan([[1,0,1],[0,2,-1],[1,-1,1]])
```

Output:

$$\begin{bmatrix} 1 & 1 & 2 \\ 0 & 0 & -1 \\ 0 & 1 & 2 \end{bmatrix}, \begin{bmatrix} 0 & 0 & -1 \\ 1 & 0 & -3 \\ 0 & 1 & 4 \end{bmatrix}$$

- *Input:*

```
rat_jordan([[1,0,0],[0,1,1],[1,1,-1]])
```

Output:

$$\begin{bmatrix} -1 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 2 \\ 0 & 1 & 0 \end{bmatrix}$$

If A is symmetric and has eigenvalues with multiple orders, the matrix P returned by `rat_jordan(A)` will contain orthogonal eigenvectors (not always of norm equal to 1); i.e., `tran(P)*P` will be a diagonal matrix where the diagonal is the square norm of the eigenvectors.

Example.

Input:

```
rat_jordan([[4,1,1],[1,4,1],[1,1,4]])
```

Output:

$$\begin{bmatrix} 1 & 2 & -1 \\ 1 & 0 & 2 \\ 1 & -2 & -1 \end{bmatrix}, \begin{bmatrix} 6 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 3 \end{bmatrix}$$

6.48.5 Jordan normal form: `jordan`

The `jordan` command finds the Jordan form of a matrix.

- `jordan` takes one mandatory and one optional argument:
 - A , a square matrix.
 - Optionally, var , a variable name.
- `jordan(A)` (in all modes but **Maple**) returns a sequence $[P, J]$ of two matrices, where the columns of P are the eigenvectors of A , J is the Jordan form of A , and

$$J = P^{-1}AP$$

`jordan(A)`, in **Maple** mode, only returns the matrix J .

- `jordan(A⟨var⟩)` returns the matrix J , as above, and assigns the matrix P to the variable var .

Examples.

- *Input (not in Maple mode):*

```
jordan([[4,1,1],[1,4,1],[1,1,4]])
```

Output:

$$\begin{bmatrix} 1 & 2 & -1 \\ 1 & 0 & 2 \\ 1 & -2 & -1 \end{bmatrix}, \begin{bmatrix} 6 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 3 \end{bmatrix}$$

- *Input:*

`jordan([[4,1,1],[1,4,1],[1,1,4]],P)`

Output:

$$\begin{bmatrix} 6 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 3 \end{bmatrix}$$

then input:

`P`

Output:

$$\begin{bmatrix} 1 & 2 & -1 \\ 1 & 0 & 2 \\ 1 & -2 & -1 \end{bmatrix}$$

If A is symmetric and has eigenvalues with multiple orders, the matrix P returned by `jordan(A)` will contain orthogonal eigenvectors (not always of norm equal to 1); i.e., `tran(P)*P` will be a diagonal matrix where the diagonal is the square norm of the eigenvectors.

Example:

Input:

`jordan([[4,1,1],[1,4,1],[1,1,4]])`

Output:

$$\begin{bmatrix} 1 & 2 & -1 \\ 1 & 0 & 2 \\ 1 & -2 & -1 \end{bmatrix}, \begin{bmatrix} 6 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 3 \end{bmatrix}$$

6.48.6 Powers of a square matrix: `matpow`

The `matpow` command finds the power of a square matrix, computed using the Jordan form.

- `matpow` command takes two arguments:

- A , a square matrix.
- n , an integer.

- `matpow(A,n)` returns A^n .

Example.

Input:

`matpow([[1,2],[2,1]],n)`

Output:

$$\begin{bmatrix} \frac{3^n+(-1)^n}{2} & \frac{3^n-(-1)^n}{2} \\ \frac{3^n-(-1)^n}{2} & \frac{3^n+(-1)^n}{2} \end{bmatrix}$$

Notice that `jordan([[1,2],[2,1]])` returns

$$\begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix}, \begin{bmatrix} 3 & 0 \\ 0 & -1 \end{bmatrix}$$

6.48.7 Characteristic polynomial: charpoly

The *characteristic polynomial* of a square matrix A is the polynomial

$$P(x) = \det(xI - A)$$

The `charpoly` command finds the characteristic polynomial of a matrix. `pcar` is a synonym for `charpoly`.

- `charpoly` takes one mandatory argument and one optional argument:
 - A , a square matrix.
 - Optionally, x , a variable name.
- `charpoly(A ⟨ x ⟩)` returns the characteristic polynomial of A . It is written as the list of its coefficients if no variable name was provided or written as an expression with respect to x if there is a second argument.

Examples.

- *Input:*

```
charpoly([[4,1,-2],[1,2,-1],[2,1,0]])
```

Output:

```
[1,-6,12,-8]
```

Hence, the characteristic polynomial of this matrix is $x^3 - 6x^2 + 12x - 8$.

- *Input:*

```
charpoly([[4,1,-2],[1,2,-1],[2,1,0]],X)
```

Output:

$$X^3 - 6X^2 + 12X - 8$$
6.48.8 Characteristic polynomial using Hessenberg algorithm: pcar_hessenberg

The `pcar_hessenberg` command finds the characteristic polynomial of a matrix. It computes the polynomial using the Hessenberg algorithm (see e.g. Henri Cohen, *A Course in Computational Algebraic Number Theory*) which is more efficient ($O(n^3)$ deterministic) if the coefficients of the matrix are in a finite field or use a finite representation like approximate numeric coefficients. Note however that this algorithm behaves badly if the coefficients are, for example, in \mathbb{Q} .

- `pcar_hessenberg` takes one mandatory argument and one optional argument:
 - A , a square matrix.
 - Optionally, x , a variable name.

- `pcar_hessenberg($A \langle x \rangle$)` returns the characteristic polynomial of A . It is written as the list of its coefficients if no variable name was provided or written as an expression with respect to x if there is a second argument.

Examples.

- *Input:*

```
pcar_hessenberg([[4,1,-2],[1,2,-1],[2,1,0]] % 37)
```

Output:

```
[1 % 37, (-6) % 37, 12 % 37, (-8) % 37]
```

- *Input:*

```
pcar_hessenberg([[4,1,-2],[1,2,-1],[2,1,0]] % 37, x)
```

Output:

```
(1 % 37) x^3 + ((-6) % 37) x^2 + (12 % 37) x + (-8) % 37
```

Hence, the characteristic polynomial of $[[4,1,-2],[1,2,-1],[2,1,0]]$ in $\mathbb{Z}/37\mathbb{Z}$ is $x^3 - 6x^2 + 12x - 8$.

6.48.9 Minimal polynomial: pmin

The minimal polynomial of a square matrix A is the polynomial P having minimal degree such that $P(A) = 0$. The `pmin` command finds the minimal polynomial of a matrix.

- `pmin` takes one mandatory argument and one optional argument:
 - A , a square matrix.
 - Optionally, x , a variable name.
- `pmin($A \langle x \rangle$)` returns the minimal polynomial A . It is written as the list of its coefficients if no variable name was provided or written as an expression with respect to x if there is a second argument.

Examples.

- *Input:*

```
pmin([[1,0],[0,1]])
```

Output:

```
[1, -1]
```

- *Input:*

```
pmin([[1,0],[0,1]], x)
```

Output:

$$x - 1$$

Hence the minimal polynomial of $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ is $x - 1$.

• *Input:*

```
pmin([[2,1,0],[0,2,0],[0,0,2]])
```

Output:

$$[1, -4, 4]$$

• *Input:*

```
pmin([[2,1,0],[0,2,0],[0,0,2]],x)
```

Output:

$$x^2 - 4x + 4$$

Hence, the minimal polynomial of $\begin{bmatrix} 2 & 1 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{bmatrix}$ is $x^2 - 4x + 4$.

6.48.10 Adjoint matrix: `adjoint_matrix`

The *comatrix* of a square matrix A of size n is the matrix B defined by $A \times B = \det(A) \times I$. The *adjoint* matrix $Q(x)$ of A is the comatrix of $xI - A$. It is a polynomial of degree $n - 1$ in x having matrix coefficients and satisfies:

$$(xI - A)Q(x) = \det(xI - A)I = P(x) \times I$$

where $P(x)$ is the characteristic polynomial of A . Since the polynomial $P(x) \times I - P(A)$ (with matrix coefficients) is also divisible by $x \times I - A$ (by algebraic identities), this means that $P(A) = 0$. We also have $Q(x) = I \times x^{n-1} + \dots + B_0$ where B_0 is the comatrix of A (times -1 if n is odd).

The `adjoint_matrix` command finds the characteristic polynomial and adjoint of a given matrix.

- `adjoint_matrix` takes one argument:
 A , a square matrix.
- `adjoint_matrix(A)` returns the list of the coefficients of $P(x)$ (the characteristic polynomial of A), and the list of the matrix coefficients of $Q(x)$ (the adjoint matrix of A).

Examples.

• *Input:*

```
adjoint_matrix([[4,1,-2],[1,2,-1],[2,1,0]])
```

Output:

$$\left[[1, -6, 12, -8], \left[\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \begin{bmatrix} -2 & 1 & -2 \\ 1 & -4 & -1 \\ 2 & 1 & -6 \end{bmatrix}, \begin{bmatrix} 1 & -2 & 3 \\ -2 & 4 & 2 \\ -3 & -2 & 7 \end{bmatrix} \right] \right]$$

Hence the characteristic polynomial is:

$$P(x) = x^3 - 6 * x^2 + 12 * x - 8$$

The determinant of A is equal to $-P(0) = 8$. The comatrix of A is equal to:

$$B = Q(0) = \begin{bmatrix} 1 & -2 & 3 \\ -2 & 4 & 2 \\ -3 & -2 & 7 \end{bmatrix}$$

Hence the inverse of A is equal to:

$$\frac{1}{8} \begin{bmatrix} 1 & -2 & 3 \\ -2 & 4 & 2 \\ -3 & -2 & 7 \end{bmatrix}$$

The adjoint matrix of A is:

$$\begin{bmatrix} x^2 - 2x + 1 & x - 2 & -2x + 3 \\ x - 2 & x^2 - 4x + 4 & -x + 2 \\ 2x - 3 & x - 2 & x^2 - 6x + 7 \end{bmatrix}$$

• *Input:*

`adjoint_matrix([[4,1],[1,2]])`

Output:

$$\left[[1, -6, 7], \left[\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} -2 & 1 \\ 1 & -4 \end{bmatrix} \right] \right]$$

Hence the characteristic polynomial P is:

$$P(x) = x^2 - 6 * x + 7$$

The determinant of A is equal to $+P(0) = 7$. The comatrix of A is equal to

$$Q(0) = - \begin{bmatrix} -2 & 1 \\ 1 & -4 \end{bmatrix}$$

Hence the inverse of A is equal to:

$$-\frac{1}{7} \begin{bmatrix} -2 & 1 \\ 1 & -4 \end{bmatrix}$$

The adjoint matrix of A is:

$$- \begin{bmatrix} x - 2 & 1 \\ 1 & x - 4 \end{bmatrix}$$

6.48.11 Companion matrix of a polynomial: companion

The `companion` command finds a matrix given its characteristic polynomial; specifically, if the polynomial is $P(x) = x^n + a_{n-1}x^{n-1} + \dots + a_{-1}x + a_0$, this matrix is equal to the identity matrix of size $n-1$ bordered with $[0, 0, \dots, 0, -a_0]$ as first row, and with $[-a_0, -a_1, \dots, -a_{n-1}]$ as last column.

- `companion` takes two arguments:
 - P , a unitary polynomial.
 - x , the name of its variable.
- `companion(P, x)` returns the matrix whose characteristic polynomial is P .

Examples.

- *Input:*

```
companion(x^2+5x-7,x)
```

Output:

$$\begin{bmatrix} 0 & 7 \\ 1 & -5 \end{bmatrix}$$

- *Input:*

```
companion(x^4+3x^3+2x^2+4x-1,x)
```

Output:

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & -4 \\ 0 & 1 & 0 & -2 \\ 0 & 0 & 1 & -3 \end{bmatrix}$$

6.48.12 Hessenberg matrix reduction: hessenberg SCHUR

A *Hessenberg* matrix is a square matrix where the coefficients below the sub-principal diagonal are all 0s. The `hessenberg` command finds a Hessenberg matrix equivalent to a given square matrix.

- `hessenberg` takes one mandatory argument and one optional argument:
 - A , a matrix.
 - n , an integer, either 0, -1 , -2 or a prime number greater than 1 (by default $n = 0$).
- `hessenberg($A \langle n \rangle$)` returns a list $[P, B]$ with $B = P^{-1}AP$ and:
 - if $n = 0$, B is a Hessenberg matrix.

- if $n = -1$, the calculations are approximate and B is upper triangular.
- if $n = -2$, the calculations are approximate, P is orthogonal and B has zero sub-subdiagonal elements.

$\text{SCHUR}(A)$ is equivalent to $\text{hessenberg}(A, -1)$, which is compatible with HP calculators.

Examples.

- *Input:*

```
A:=[ [3,2,2,2,2], [2,1,2,-1,-1], [2,2,1,-1,1], [2,-1,-1,3,1], [2,-1,1,1,2] ];
[P,B]:=hessenberg(A)
```

or:

```
[P,B]:= hessenberg(A,0);
```

Output:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & \frac{1}{2} & \frac{1}{4} & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}, \begin{bmatrix} 3 & 8 & 5 & \frac{5}{2} & 2 \\ 2 & 1 & \frac{1}{2} & -\frac{5}{4} & -1 \\ 0 & 2 & 1 & 2 & 0 \\ 0 & 0 & 2 & \frac{3}{2} & 2 \\ 0 & 0 & 0 & \frac{13}{8} & \frac{7}{2} \end{bmatrix}$$

Indeed:

Input:

```
pcar(A)
```

and:

```
pcar(B)
```

both return:

Output:

```
[1, -10, 13, 71, -50, -113]
```

and it is easily verified that $B=\text{inv}(P)*A*P$.

- With A as above: *Input:*

```
B:= hessenberg(A,-1);
```

Output (to 2 digits):

$$\begin{bmatrix} \begin{bmatrix} 0.73 & -0.057 & -0.42 & -0.17 & -0.51 \\ 0.25 & -0.53 & 0.72 & -0.38 & -0.048 \\ 0.35 & -0.44 & -0.3 & 0.19 & 0.74 \\ 0.34 & 0.68 & 0.17 & -0.46 & 0.43 \\ 0.41 & 0.25 & 0.44 & 0.76 & -0.063 \end{bmatrix} \end{bmatrix},$$

$$\begin{bmatrix} 6.7 & 8.7e-15 & -2e-13 & 2.7e-14 & -1.4e-13 \\ 0.0 & 4.6 & 0 & 0 & 0 \\ 0.0 & 0.0 & -1.9 & 0 & 0 \\ 0.0 & 0.0 & 0.0 & 1.7 & 0 \\ 0.0 & 0.0 & 0.0 & -0.0 & -1.2 \end{bmatrix}$$

6.48.13 Hermite normal form: ihermite

The *Hermite normal form* of a matrix A with integer coefficients is a sort of integer row-echelon form. It is an upper triangular matrix B such that $B = UA$ for a matrix U which is invertible in \mathbb{Z} ($\det(U) = \pm 1$). The `ihermite` command finds the Hermite normal form of a matrix.

- `ihermite` takes one argument:
 A , a matrix with coefficients in \mathbb{Z} .
- `ihermite(A)` return a list $[U, B]$ as above, and the absolute value of the elements above the diagonal of B are less than the pivot of the column divided by 2.

The result is obtained by a Gauss-like reduction algorithm using only operations of rows with integer coefficients and invertible in \mathbb{Z} .

Example.

Input:

```
A:=[[9,-36,30],[-36,192,-180],[30,-180,180]]:; ihermite(A)
```

Output:

$$\begin{bmatrix} 13 & 9 & 7 \\ 6 & 4 & 3 \\ 20 & 15 & 12 \end{bmatrix}, \begin{bmatrix} 3 & 0 & 30 \\ 0 & 12 & 0 \\ 0 & 0 & 60 \end{bmatrix}$$

Application: Compute a \mathbb{Z} -basis of the kernel of a matrix having integer coefficients

Let M be a matrix with integer coefficients. To find the nullspace of M , you want find what you can multiply M by on the right to get the zero vector, but the Hermite command returns a matrix that you multiply on the left of M . So consider the transpose of M , M^T .

Let A be the Hermite normal form of M^T , and U an invertible matrix in \mathbb{Z} such that $A = UM^T$. Transposing this, you'll get $A^T = MU^T$. Note that M times a column of U^T equals the corresponding column of A^T . So if a column of A^T is the zero vector, then M times the corresponding column of U^T will be the zero vector. In fact, these columns of U^T will be a \mathbb{Z} -basis for the nullspace of M .

Any columns of A^T which are all 0s correspond to the rows of A which are all 0s. Since A is in Hermite form, these will be at the bottom, and so the corresponding rows of U will be at the bottom, and will be a \mathbb{Z} -basis for the nullspace of M .

As an example, consider the matrix M :

Input:

```
M:=[[1,4,7],[2,5,8],[3,6,9]]
```

Find the Hermite decomposition:

Input:

```
(U,A):=ihermite(transpose(M))
```

Output:

$$\begin{bmatrix} -3 & 1 & 0 \\ 4 & -1 & 0 \\ -1 & 2 & -1 \end{bmatrix}, \begin{bmatrix} 1 & -1 & -3 \\ 0 & 3 & 6 \\ 0 & 0 & 0 \end{bmatrix}$$

Only the third row of **A** consists of all 0s, so a \mathbb{Z} -basis for the nullspace of **M** consists of only the third row of **U**; namely $\mathbf{U}[2]=[-1, 2, -1]$.

You can check that this is in the nullspace:

Input:

```
M*U[2]
```

Output:

```
[0,0,0]
```

6.48.14 Smith normal form in \mathbb{Z} : `ismith`

A matrix B is in *Smith normal form* if the only non-zero entries are on the diagonal (for non-square matrices, this simply means that $b_{ij} = 0$ for $i \neq j$) and $b_{i,i}$ divides $b_{i+1,i+1}$. The elements $b_{i,i}$ are called invariant factors and are used to describe the structure of finite abelian groups.

For any matrix A with coefficients in \mathbb{Z} , there exist matrices U and V , invertible in \mathbb{Z} , such that $B = UAV$ is in Smith normal form and has coefficients in \mathbb{Z} . The `ismith` command finds the matrices U , B and V .

- `ismith` takes one argument:
 A , a matrix with coefficients in \mathbb{Z} .
- `ismith(A)` returns a list $[U, B, V]$ of three matrices such that $B = UAV$ is in Smith normal form and U and V are invertible in \mathbb{Z} .

Example.

Input:

```
A:=[[9,-36,30],[-36,192,-180],[30,-180,180]];;
U,B,V:=ismith(A)
```

Output:

$$\begin{bmatrix} -3 & 0 & 1 \\ 6 & 4 & 3 \\ 20 & 15 & 12 \end{bmatrix}, \begin{bmatrix} 3 & 0 & 0 \\ 0 & 12 & 0 \\ 0 & 0 & 60 \end{bmatrix}, \begin{bmatrix} 1 & 24 & -30 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The invariant factors are 3, 12 and 60.

6.48.15 Smith normal form: smith

The **smith** command finds the Smith normal form of a matrix with elements in a field K .

- **smith** takes one argument:
 A , a square matrix with elements in a field K .
- **smith**(A) returns a list $[U, V, D]$ of three matrices where U and V are invertible, D is diagonal, and $D = UAV$. Input:

Examples.

- *Input:*

```
M:=( [ [5,-2,3,6], [1,-3,1,3], [7,-6,-4,7], [-2,-4,-3,0]] ) % 17 ;;
A:= x*idn(4) -M
```

Output:

$$\begin{pmatrix} x + (-5) \% 17 & 2 \% 17 & (-3) \% 17 & (-6) \% 17 \\ (-1) \% 17 & x + 3 \% 17 & (-1) \% 17 & (-3) \% 17 \\ (-7) \% 17 & 6 \% 17 & x + 4 \% 17 & (-7) \% 17 \\ 2 \% 17 & 4 \% 17 & 3 \% 17 & x \end{pmatrix}$$

Input:

```
U, D, V:= smith(A);
U
```

Output:

$$\begin{bmatrix} 0 \% 17 & (-1) \% 17 & 0 \% 17 & 0 \% 17 \\ 0 \% 17 & 0 \% 17 & 6 \% 17 & 4 \% 17 \\ (-2x + 5) \% 17 & (-4x - 5) \% 17 & (-3x - 6) \% 17 & (x^2 - 3x + 6) \% 17 \\ (2x^2 + 5x + 6) \% 17 & (4x^2 + 8x + 2) \% 17 & (3x^2 + 4x + 1) \% 17 & (-x^3 - 2x^2 + 2x - 6) \% 17 \end{bmatrix}$$

Input:

V

Output:

$$\begin{bmatrix} 1 \% 17 & (x + 3) \% 17 & (-6x^2 - 3x - 7) \% 17 & (6x^5 + 2x^4 - 2x^3 + x^2 - 8x + 6) \% 17 \\ 0 \% 17 & 1 \% 17 & (-6x - 2) \% 17 & (6x^4 + x^3 - 6x^2 + 5x - 6) \% 17 \\ 0 \% 17 & 0 \% 17 & 1 \% 17 & (-x^3 + 3x^2 + 7) \% 17 \\ 0 \% 17 & 0 \% 17 & 0 \% 17 & 1 \% 17 \end{bmatrix}$$

Input:

D

Output:

$$\begin{bmatrix} 1 \% 17 & 0 \% 17 & 0 \% 17 & 0 \% 17 \\ 0 \% 17 & 1 \% 17 & 0 \% 17 & 0 \% 17 \\ 0 \% 17 & 0 \% 17 & 1 \% 17 & 0 \% 17 \\ 0 \% 17 & 0 \% 17 & 0 \% 17 & (-x^4 - 2x^3 + 8x^2 - 3x + 2) \% 17 \end{bmatrix}$$

You can check this:

Input:

`normal(U*A*V-D)`

Output:

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

- *Input:*

```
B:=[x^2+x-1,1,0,1],[-1,x,0,-1],[0,x^2+1,x,0],[1,0,1,x^2+x+1]] % 3;;
L:=smith(B)
```

Output:

$$\begin{bmatrix} 0 \% 3 & (-1) \% 3 & 0 \% 3 & 0 \% 3 \\ 1 \% 3 & 0 \% 3 & 0 \% 3 & (-x^2 - x + 1) \% 3 \\ 0 \% 3 & (x^2 + 1) \% 3 & (-x) \% 3 & (x^2 + 1) \% 3 \\ (-1) \% 3 & (-x^4 - x^3 + x + 1) \% 3 & (x^3 + x^2 - x + 1) \% 3 & (-x^4 - x^3 + x^2 - x) \% 3 \end{bmatrix}$$

6.49 Matrix factorizations

Note that most matrix factorization algorithms are implemented numerically, only a few of them will work symbolically.

6.49.1 Cholesky decomposition: `cholesky`

If M is a square symmetric positive definite matrix, the Cholesky decomposition is $M = P^T P$, where P is a lower triangular matrix. The `cholesky` command finds the matrix P .

- `cholesky` takes one argument:
 M , a square symmetric positive definite matrix.
- `cholesky(M)` returns a symbolic or numeric matrix P given by the Cholesky decomposition.

Examples.

- *Input:*

```
cholesky([[1,1],[1,5]])
```

Output:

$$\begin{bmatrix} 1 & 0 \\ 1 & 2 \end{bmatrix}$$

- *Input:*

```
cholesky([[3,1],[1,4]])
```

Output:

$$\begin{bmatrix} \sqrt{3} & 0 \\ \frac{\sqrt{3}}{3} & \frac{\sqrt{33}}{3} \end{bmatrix}$$

- *Input:*

```
cholesky([[1,1],[1,4]])
```

Output:

$$\begin{bmatrix} 1 & 0 \\ 1 & \sqrt{3} \end{bmatrix}$$

Warning: If the matrix argument A is not a symmetric matrix, `cholesky(A)` does not return an error, instead `cholesky(A)` will use the symmetric matrix B of the the quadratic form q corresponding to the (non symmetric) bilinear form of the matrix A .

Example.

Input:

```
cholesky([[1,-1],[-1,4]])
```

or:

```
cholesky([[1,-3],[1,4]])
```

Output:

$$\begin{bmatrix} 1 & 0 \\ -1 & \sqrt{3} \end{bmatrix}$$

6.49.2 QR decomposition: qr

The QR decomposition of a square matrix A is $A = QR$, where Q is an orthogonal matrix ($Q^T Q = I$) and R is upper triangular. The `qr` command finds the QR decomposition of a matrix.

- `qr` takes one argument:
 A , a numeric square matrix.
- `qr(A)` returns a list $[Q, R]$ with Q and R from the QR decomposition.

Examples.

- *Input:*

```
A := [[3,5],[4,5]];;
qr(A)
```

Output:

$$\begin{pmatrix} \frac{3}{5} & \frac{4}{5} \\ \frac{4}{5} & -\frac{3}{5} \end{pmatrix}, \begin{bmatrix} 5 & 7 \\ 0 & 1 \end{bmatrix}$$

- *Input:*

```
qr([[1,2],[3,4]])
```

Output:

$$\begin{pmatrix} \frac{1}{\sqrt{10}} & \frac{3}{5\sqrt{10}} \\ \frac{3}{\sqrt{10}} & -\frac{1}{5\sqrt{10}} \end{pmatrix}, \begin{bmatrix} \sqrt{10} & \frac{7}{5}\sqrt{10} \\ 0 & \frac{\sqrt{10}}{5} \end{bmatrix}$$

6.49.3 QR decomposition (for TI compatibility): QR

The QR command finds the QR decomposition of a matrix.

- QR takes three arguments:
 - A , a square matrix.
 - q and r , two variable names.
- $\text{QR}(A, q, r)$ returns the matrix R from the QR decomposition of A , and assigns the matrices Q and R to the variables q and r .

Example.

Input:

```
QR([[3,5],[4,5]],Q,R)
```

Output (the matrix R):

$$\begin{bmatrix} 5 & 7 \\ 0 & 1 \end{bmatrix}$$

Input:

Q

Output (the matrix Q):

$$\begin{pmatrix} \frac{3}{5} & \frac{4}{5} \\ \frac{4}{5} & -\frac{3}{5} \end{pmatrix}$$

6.49.4 LQ decomposition (HP compatible): LQ

The LQ decomposition of a matrix A is $A = LQP$, where L is lower triangular the same size as A (if A is not square, then $\ell_{i,j} = 0$ for $i > j$), Q is an orthogonal matrix, and P is a permutation matrix. The `LQ` command finds the LQ decomposition of a matrix.

- `LQ` takes one argument:
 A , a matrix.
- `LQ(A)` returns a list $[L, Q, P]$ of the matrices given by the LQ decomposition.

Examples.

- *Input:*

$$L, Q, P := \text{LQ}([[4, 0, 0], [8, -4, 3]])$$

Output:

$$\left[\begin{pmatrix} 4.0 & 0.0 & 0.0 \\ 8.0 & 5.0 & -4.4408920985 \times 10^{-16} \end{pmatrix}, \begin{pmatrix} 1.0 & 0.0 & 0.0 \\ 0.0 & -0.8 & 0.6 \\ 0.0 & -0.6 & -0.8 \end{pmatrix}, \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \right]$$

Here, $L*Q$ is the same as $P*A$.

- *Input:*

$$L, Q, P := \text{LQ}([[24, 18], [30, 24]])$$

Output:

$$\left[\begin{pmatrix} -30.0 & 0.0 \\ -38.4 & -1.2 \end{pmatrix}, \begin{pmatrix} -0.8 & -0.6 \\ 0.6 & -0.8 \end{pmatrix}, \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \right]$$

Again, $L*Q = P*A$.

6.49.5 LU decomposition: lu

The LU decomposition of a square matrix A is $PA = LU$, where P is a permutation matrix, L is lower triangular with 1s on the diagonal, and U is upper triangular. The `lu` command finds the LU decomposition of a matrix.

- `lu` takes one argument:
 A , a square matrix.
- `lu(A)` returns a list $[p, L, U]$ where p is a permutation that determines P , and P, L and U are the LU decomposition of A .

The permutation matrix P is defined from p by:

$$P_{i,p(i)} = 1, \quad P_{i,j} = 0 \text{ if } j \neq p(i)$$

In other words, it is the identity matrix where the rows are permuted according to the permutation p . You can get the permutation matrix from p by $P := \text{permu2mat}(p)$ (see Section 6.9.6 p.192).

Example.

Input:

```
A := [[3.,5.],[4.,5.]];;
(p,L,U):=lu(A)
```

Output:

$$[1,0], \begin{bmatrix} 1 & 0 \\ 0.75 & 1 \end{bmatrix}, \begin{bmatrix} 4.0 & 5.0 \\ 0 & 1.25 \end{bmatrix}$$

Here $n = 2$, hence:

$$P[0,p(0)] = P_2[0,1] = 1, \quad P[1,p(1)] = P_2[1,0] = 1, \quad P = [[0,1],[1,0]]$$

Verification:

Input:

```
permu2mat(p)*A; L*U
```

Output:

$$\begin{bmatrix} 4.0 & 5.0 \\ 3.0 & 5.0 \end{bmatrix}, \begin{bmatrix} 4.0 & 5.0 \\ 3.0 & 5.0 \end{bmatrix}$$

Note that the permutation is different for exact input (the choice of pivot is the simplest instead of the largest in absolute value).

Input:

```
lu([[1,2],[3,4]])
```

Output:

$$[0,1], \begin{bmatrix} 1 & 0 \\ 3 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 2 \\ 0 & -2 \end{bmatrix}$$

Input:

```
lu([1.0,2],[3,4]))
```

Output:

$$[1,0], \begin{bmatrix} 1 & 0 \\ 0.333333333333 & 1 \end{bmatrix}, \begin{bmatrix} 3.0 & 4.0 \\ 0 & 0.666666666667 \end{bmatrix}$$

6.49.6 LU decomposition (for TI compatibility): LU

The LU command finds the LU decomposition of a matrix.

- LU takes four arguments:
 - A , a numeric square matrix.
 - l, u and p , three variable names.
- LU(A) returns the matrix P from the LU decomposition of A , and assigns L, U and P to the variables l, u and p . Namely, P is a permutation matrix, L is lower triangular with 1s on the diagonal, and U is upper triangular with $PA = LU$.

Example.

Input:

LU([[3,5],[4,5]],L,U,P)

Output:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

[[0,1],[1,0]]

Input:

L

Output:

$$\begin{bmatrix} 1 & 0 \\ \frac{4}{3} & 1 \end{bmatrix}$$

Input:

U

Output:

$$\begin{bmatrix} 3 & 5 \\ 0 & -\frac{5}{3} \end{bmatrix}$$

Input:

P

Output:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

6.49.7 Singular values (HP compatible): SVL svl

The singular values of a matrix A are the positive square roots of the eigenvalues of $A \cdot A^T$. So, if A is symmetric, the singular values are the absolute values of the eigenvalues of A . The SVL (or `svl`) command finds the singular values of a matrix.

`svl` is a synonym for `SVL`.

- SVL takes one argument:
 A , a matrix.
- `SVL(A)` returns a list of the singular values of A .

Examples.

- *Input:*

```
SVL([[1,2],[3,4]])
```

or:

```
svl([[1,2],[3,4]])
```

Output:

```
[0.365966190626, 5.46498570422]
```

- *Input:*

```
evalf(sqrt(eigenvals([[1,2],[3,4]]*transpose([[1,2],[3,4]]))))
```

Output:

```
5.46498570422, 0.365966190626
```

- *Input:*

```
SVL([[1,4],[4,1]])
```

or:

```
svl([[1,4],[4,1]])
```

Output:

```
[5.0, 3.0]
```

Input:

```
abs(eigenvals([[1,4],[4,1]]))
```

Output:

```
5, 3
```

6.49.8 Singular value decomposition: svd

The *singular value decomposition* of a matrix A is a factorization $A = USQ^T$, where U and Q are orthogonal and S is a diagonal matrix. The `svd` command finds the singular value decomposition of a matrix.

- `svd` takes one argument:
 A , a numeric square matrix.
- `svd(A)` returns a list $[U, s, Q]$ where U and Q are the orthogonal matrices of the singular value decomposition and s is the diagonal of the matrix S .

You can get the diagonal matrix S from s with $S = \text{diag}(s)$ (see Section 6.44.2 p.503).

Examples.

- *Input:*

$$\text{svd}([[1, 2], [3, 4]])$$

Output:

$$\begin{bmatrix} -0.404553584834 & -0.914514295677 \\ -0.914514295677 & 0.404553584834 \end{bmatrix}, [5.46498570422, 0.365966190626], \begin{bmatrix} -0.576048436766 & 0 \\ -0.81741556047 & -0 \end{bmatrix}$$

- *Input:*

$$(U, s, Q) := \text{svd}([[3, 5], [4, 5]])$$

Output:

$$\begin{bmatrix} -0.672988041811 & -0.739653361771 \\ -0.739653361771 & 0.672988041811 \end{bmatrix}, [8.6409011028, 0.578643354497], \begin{bmatrix} -0.576048436766 & 0 \\ -0.81741556047 & -0 \end{bmatrix}$$

Verification:

Input:

$$U * \text{diag}(s) * \text{tran}(Q)$$

Output:

$$\begin{pmatrix} 3.0 & 5.0 \\ 4.0 & 5.0 \end{pmatrix}$$
6.49.9 Short basis of a lattice: lll

The `lll` command finds a short basis for the \mathbb{Z} -modules generated by the rows of a matrix.

- `lll` takes one argument:
 M , an invertible matrix with integer coefficients.

- $\text{lll}(M)$ returns the sequence (S, A, L, O) where:
 - the rows of S is a short basis of the \mathbb{Z} -module generated by the rows of M ,
 - A is the change-of-basis matrix from the short basis to the basis defined by the rows of M ($AM = S$),
 - L is a lower triangular matrix, the modulus of its non diagonal coefficients are less than $1/2$,
 - O is a matrix with orthogonal rows such that $L * O = S$.

Examples.

- *Input:*

$$(S, A, L, O) := \text{lll}(M := [[2, 1], [1, 2]])$$

Output:

$$\begin{bmatrix} -1 & 1 \\ 2 & 1 \end{bmatrix}, \begin{bmatrix} -1 & 1 \\ 1 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ -\frac{1}{2} & 1 \end{bmatrix}, \begin{bmatrix} -1 & 1 \\ \frac{3}{2} & \frac{3}{2} \end{bmatrix}$$

$$[[[-1, 1], [2, 1]], [[-1, 1], [1, 0]], [[1, 0], [1/-2, 1]],$$

$$[[[-1, 1], [3/2, 3/2]]]$$

Hence:

$$S = \begin{bmatrix} -1 & 1 \\ 2 & 1 \end{bmatrix}$$

$$A = \begin{bmatrix} -1 & 1 \\ 1 & 0 \end{bmatrix}$$

$$L = \begin{bmatrix} 1 & 0 \\ -\frac{1}{2} & 1 \end{bmatrix}$$

$$O = \begin{bmatrix} -1 & 1 \\ \frac{3}{2} & \frac{3}{2} \end{bmatrix}$$

So the original basis is $v_1 = [2, 1], v_2 = [1, 2]$ and the short basis is $w_1 = [-1, 1], w_2 = [2, 1]$. Since $w_1 = -v_1 + v_2$ and $w_2 = v_1$ then $AM = SLO = S$.

- *Input:*

$$M := [[3, 2, 1], [1, 2, 3], [2, 3, 1]] ;$$

$$(S, A, L, O) := \text{lll}(M)$$

Output:

$$\begin{bmatrix} -1 & 1 & 0 \\ -1 & -1 & 2 \\ 3 & 2 & 1 \end{bmatrix}, \begin{bmatrix} -1 & 0 & 1 \\ 0 & 1 & -1 \\ 1 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -\frac{1}{2} & -\frac{1}{2} & 1 \end{bmatrix}, \begin{bmatrix} -1 & 1 & 0 \\ -1 & -1 & 2 \\ 2 & 2 & 2 \end{bmatrix}$$

so

$$S = \begin{bmatrix} -1 & 1 & 0 \\ -1 & -1 & 2 \\ 3 & 2 & 1 \end{bmatrix}$$

$$A = \begin{bmatrix} -1 & 0 & 1 \\ 0 & 1 & -1 \\ 1 & 0 & 0 \end{bmatrix}$$

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -\frac{1}{2} & -\frac{1}{2} & 1 \end{bmatrix}$$

$$O = \begin{bmatrix} -1 & 1 & 0 \\ -1 & -1 & 2 \\ 2 & 2 & 2 \end{bmatrix}$$

Properties:

$AM = S$ and $LO = S$.

6.50 Different matrix norms

See Section 6.42.1 p.493 for different norms on vectors.

6.50.1 The Frobenius norm: `frobenius_norm`

The *Frobenius norm* of a matrix A is $\sqrt{\sum_{i,j} a_{i,j}^2}$. The `frobenius_norm` command finds the Frobenius norm of a matrix.

- `frobenius_norm` takes one argument:
 A , a matrix.
- `frobenius_norm(A)` returns the Frobenius norm of A .

Example.

Input:

```
frobenius_norm([[1,2,3],[3,-9,6],[4,5,6]])
```

Output:

$$\sqrt{217}$$

since $\sqrt{1^2 + 2^2 + 3^2 + 3^2 + (-9)^2 + 6^2 + 4^2 + 5^2 + 6^2} = \sqrt{217}$.

6.50.2 ℓ^2 matrix norm: `norm` `l2norm`

The ℓ^2 norm of a matrix is an operator norm (see Section 6.50.6 p.575) induced by the ℓ^2 norm on vectors (see Section 6.42.1 p.493). The `l2norm` computes the ℓ^2 norm of a matrix.

`norm` is a synonym for `l2norm`.

- `l2norm` takes one argument:
 A , a matrix.

- `l2norm(A)` returns the ℓ^2 norm of A .

Example.

Input:

```
l2norm([[1,2],[3,-4]])
```

Output:

```
5.11667273602
```

6.50.3 ℓ^∞ matrix norm: `maxnorm`

The ℓ^∞ norm of a matrix A is $\max_{j,k}(|a_{j,k}|)$. The `maxnorm` command finds the ℓ^∞ norm of a matrix. (See also Section 6.42.1 p.493.)

- `maxnorm` takes one argument:
 A , a matrix.
- `maxnorm(A)` returns the ℓ^∞ norm of A .

Example.

Input:

```
maxnorm([[1,2],[3,-4]])
```

Output:

```
4
```

6.50.4 Matrix row norm: `rownorm` `rowNorm`

The row norm of a matrix A is $\max_k(\sum_j |a_{j,k}|)$. (This is also an operator norm; see Section 6.50.6 p.575.) The `rownorm` command finds the row norm of a matrix.

`rowNorm` is a synonym for `rownorm`, and for matrices `linfnorm` is also a synonym.

- `rownorm` takes one argument:
 A , a matrix
- `rownorm(A)` returns the row norm of A .

Example.

Input:

```
rownorm([[1,2],[3,-4]])
```

Output:

```
7
```

Indeed: $\max(1+2, 3+4) = 7$.

6.50.5 Matrix column norm: `colnorm` `colNorm` `l1norm`

The column norm of a matrix A is $\max_j(\sum_k |a_{j,k}|)$. (This is also an operator norm; see Section 6.50.6 p.575.) The `colnorm` command finds the column norm of a matrix.

`colNorm` is a synonym for `colnorm`, and for matrices `l1norm` is also a synonym.

- `colnorm` takes one argument:
 A , a matrix
- `colnorm(A)` returns the column norm of A .

Example:

Input:

```
colnorm([[1,2],[3,-4]])
```

Output:

6

Indeed: $\max(1+3, 2+4) = 6$

6.50.6 The operator norm of a matrix: `matrix_norm` `l1norm` `l2norm` `norm` `specnorm` `linfnorm`**Operator norms**

In mathematics, particularly functional analysis, a linear function between two normed spaces $f : E \rightarrow F$ is continuous exactly when there is a number K such that $\|f(x)\|_F \leq K\|x\|$ for all x in E . (See Section 6.42.1 p.493 for norms on \mathbb{R}^n .) For this reason, they are also called bounded linear functions. The infimum of all such K is defined to be the operator norm of f , and it depends on the norms of E and F . There are other characterizations of the operator norm of f , such as the supremum of $\|f(x)\|_F$ over all x in E with $\|x\|_E \leq 1$.

If E and F are finite dimensional, then any linear function $f : E \rightarrow F$ will be bounded.

Any $m \times n$ matrix $A = (a_{jk})$ corresponds to a linear function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ defined by $f(x) = Ax$. The operator norm of A will be the operator norm of f .

- If \mathbb{R}^n and \mathbb{R}^m both have the ℓ_1 norm, namely for $x = (x_1, x_2, \dots)$ the norm is $\|x\| = \sum_j |x_j|$, the operator norm of A is

$$\max_k \left(\sum_j |a_{jk}| \right).$$

This is the column norm given by `colnorm(A)` (see Section 6.50.5 p.575).

- If \mathbb{R}^n and \mathbb{R}^m both have the ℓ_2 norm, namely for $x = (x_1, x_2, \dots)$ the norm is $\|x\| = \sqrt{\sum_j x_j^2}$ (the usual Euclidean norm), the operator norm of A is the largest eigenvalue of $f^* \circ f$, where f^* is the transpose of f , and so the largest singular value of f . This is given by `l2norm` (see Section 6.50.2 p.573) or `max(SVL(A))` (see Section 6.49.7 p.570).
- If \mathbb{R}^n and \mathbb{R}^m both have the ℓ_∞ norm, namely for $x = (x_1, x_2, \dots)$ the norm is $|x| = \max_j |x_j|$, the operator norm of A is

$$\max_j \left(\sum_k |a_{jk}| \right).$$

This is given by `rownorm(A)` (see Section 6.50.4 p.574).

Computing operator norms

The `matrix_norm` command is a command which can find any of the above operator norms.

- `matrix_norm` takes two arguments:
 - A , a matrix.
 - arg , which can be 1, 2 or `inf`.
- `matrix_norm(A, arg)` returns an operator norm of the operator associated to the matrix, the norm is determined by arg .
 - If arg is 1, it is based on the ℓ^1 norm on \mathbb{R}^n .
`matrix_norm(A, 1)` is the same as `colnorm(A)` and `l1norm(A)`.
 - If arg is 2, it is based on the ℓ^2 norm on \mathbb{R}^n .
`matrix_norm(A, 2)` is the same as `l2norm(A)` and `norm(A)`.
 - If arg is `inf`, it is based on the ℓ^∞ norm on \mathbb{R}^n .
`matrix_norm(A, inf)` is the same as `rownorm(A)` and `linfnorm(A)`.

Examples.

- *Input:*

```
B:= [[1,2,3],[3,-9,6],[4,5,6]]
```

then:

```
matrix_norm(B,1)
```

or:

```
l1norm(B)
```

or:

```
colnorm(B)
```


Output:

16

since $\max(1 + 3 + 4, 2 + 9 + 5, 3 + 6 + 6) = 16$.

- *Input:*

`matrix_norm(B,2)`

or:

`l2norm(B)`

Output:

11.2449175989

- *Input:*

`matrix_norm(B,inf)`

or:

`linfnorm(B)`

or:

`rowNorm(B)`

Output:

18

since $\max(1 + 2 + 3, 3 + 9 + 6, 4 + 5 + 6) = 18$.

6.51 Isometries

An isometry of \mathbb{R}^n is a distance-preserving map. In \mathbb{R}^2 , the isometries are made up of:

- reflection across a line.
- rotation about a point.
- translation.

In \mathbb{R}^3 , the isometries are made up of:

- reflection across a plane.
- rotation about a line.
- translation.

An isometry is direct if it preserves orientation (it doesn't involve a reflection), otherwise it is indirect.

An $n \times n$ matrix A determines an isometry of the function $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$ by $f(\mathbf{x}) = A\mathbf{x}$ is an isometry. Such isometries fix the origin, so they can't involve translation, can only rotate about the origin, and the line or plane of reflection will pass through the origin.

An isometry in \mathbb{R}^2 can be characterized by:

- The angle of rotation (for a direct isometry) or a normal to the line of reflection (for an indirect isometry).
- +1 or -1 to indicate whether it is direct or indirect; +1 for direct and -1 for indirect.

An isometry in \mathbb{R}^3 can be characterized by:

- The direction of an axis of rotation.
- The angle of rotation (for a direct isometry) or a normal to the plane of reflection (for an indirect isometry).
- +1 or -1 to indicate whether it is direct or indirect; +1 for direct and -1 for indirect.

6.51.1 Recognizing an isometry: `isom`

The `isom` command determines whether or not a 2×2 or 3×3 matrix determines an isometry, and if it does, finds a characterization.

- `isom` takes one argument:
 A , a 2×2 or 3×3 matrix.
- `isom(A)` returns `[0]` if A does not determine an isometry, otherwise it returns a list `[char, n]`, where *char* is the characteristic element of a list of characteristic elements and *n* is 1 for a direct isometry and -1 for an indirect isometry.
 - For a 2×2 matrix, *char* is the angle of rotation about the origin for a direct isometry or a vector determining the line (through the origin) of reflection for an indirect symmetry.
 - For a 3×3 matrix, *char* is a list consisting of the axis direction and angle of rotation for a direct isometry or a vector normal to the plane of reflection for an indirect isometry.

Examples.

- *Input:*

```
isom([[0,0,1],[0,1,0],[1,0,0]])
```

Output:

$$[[1, 0, -1], -1]$$

which means that this isometry is a 3-d symmetry with respect to the plane $x - z = 0$.

- *Input:*

$$\text{isom}(\sqrt{2}/2*[[1, -1], [1, 1]])$$

Output:

$$\left[\frac{\pi}{4}, 1\right]$$

Hence, this isometry is a 2-d rotation of angle $\frac{\pi}{4}$.

- *Input:*

$$\text{isom}([[0, 0, 1], [0, 1, 0], [0, 0, 1]])$$

Output:

$$[0]$$

therefore this transformation is not an isometry.

6.51.2 Finding the matrix of an isometry: `mkisom`

The `mkisom` command finds the matrix of an isometry given the characteristic elements.

- `mkisom` takes two arguments:
 - *char*, the characteristic element (for isometries of \mathbb{R}^2 or a list of characteristic elements (for isometries of \mathbb{R}^3 .
 For isometries of \mathbb{R}^2 , *char* will be the angle of rotation for direct isometries or a vector determining the line (through the origin) of reflection for an indirect symmetry.
 For isometries of \mathbb{R}^3 , *char* will be the list consisting of the axis direction and angle of rotation for a direct isometry or a vector normal to the plane of reflection for an indirect isometry.
 - *n*, either +1 for a direct isometry or -1 an indirect isometry.
- `mkisom(char, n)` returns a matrix of the corresponding isometry.

Examples.

- *Input:*

$$\text{mkisom}([[-1, 2, -1], \pi], 1)$$

Output (the matrix of the rotation about axis $[-1, 2, -1]$ of angle π):

$$\begin{bmatrix} -\frac{2}{3} & -\frac{2}{3} & \frac{1}{3} \\ -\frac{2}{3} & \frac{1}{3} & -\frac{2}{3} \\ \frac{1}{3} & -\frac{2}{3} & -\frac{2}{3} \end{bmatrix}$$

- Input:

`mkisom([pi], -1)`

Output (the matrix of the symmetry with respect to O):

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

- Input:

`mkisom([1, 1, 1], -1)`

Output (the matrix of the symmetry with respect to the plane $x + y + z = 0$):

$$\begin{bmatrix} \frac{1}{3} & -\frac{2}{3} & -\frac{2}{3} \\ -\frac{2}{3} & \frac{1}{3} & -\frac{2}{3} \\ -\frac{2}{3} & -\frac{2}{3} & \frac{1}{3} \end{bmatrix}$$

- Input:

`mkisom([1, 1, 1], pi/3, -1)`

Output (the matrix of the product of a rotation of axis $[1, 1, 1]$ and angle $\frac{\pi}{3}$ and of a symmetry with respect to the plane $x + y + z = 0$):

$$\begin{bmatrix} 0 & -1 & 0 \\ 0 & 0 & -1 \\ -1 & 0 & 0 \end{bmatrix}$$

- Input:

`mkisom(pi/2, 1)`

Output (the matrix of the plane rotation of angle $\frac{\pi}{2}$):

$$\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

- Input:

`mkisom([1, 2], -1)`

Output (matrix of the plane symmetry with respect to the line of equation $x + 2y = 0$):

$$\begin{bmatrix} \frac{3}{5} & -\frac{4}{5} \\ -\frac{4}{5} & -\frac{3}{5} \end{bmatrix}$$

6.52 Linear Programming

Linear programming problems involve maximizing a linear functional under linear equality or inequality constraints. The simplest case can be solved directly by the so-called simplex algorithm. Most cases require you to solve an auxiliary linear programming problem to find an initial vertex for the simplex algorithm.

6.52.1 Simplex algorithm: `simplex_reduce`

The simple case

The simplest linear programming problem is to find the maximum of an objective function

$$z(x_1, \dots, x_n) = c_1x_1 + \dots + c_nx_n$$

under the constraints (where $b_i \geq 0, i = 1, \dots, m$):

$$a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n \leq b_1 \quad (6.7)$$

$$\vdots \quad \quad \quad \vdots \quad (6.8)$$

$$a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n \leq b_m \quad (6.9)$$

$$(6.10)$$

and

$$x_i \geq 0 \quad i = 1, \dots, n.$$

This can be abbreviated

$$\max(c \cdot x), \quad Ax \leq b, \quad x \geq 0, \quad b \geq 0$$

(where the vector inequalities mean term-by-term inequalities). The constraints determine a simplex in \mathbb{R}^n , and the standard method to solve this problem is called the *simplex method*.

The simplex method

Xcas can do the simplex method for you, but it can still help to get a rough idea of how it works. You can google “simplex method” for more details.

The constraint inequalities (6.7) can be turned into equalities by adding an auxiliary (surplus) variable for each inequality. The constraints are equivalent to:

$$a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n + s_1 + \dots + 0 = b_1$$

$$\vdots \quad \quad \quad \vdots$$

$$a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n + 0 + \dots + s_m = b_m$$

for some $s_1, \dots, s_m \geq 0$. The straightforward solution to this system of equations, namely $x_i = 0$ for $i = 1, \dots, n$ and $s_j = b_j$ for $j = 1, \dots, m$, will

be the beginning solution. The objective function

$$\begin{aligned} z &= c_1x_1 + \cdots + c_nx_n \\ &= c_1x_1 + \cdots + c_nx_n + 0s_1 + \cdots + 0s_m \end{aligned}$$

will be zero at this solution. If one of the coefficients of the objective function is positive, you can find a larger value of x_i that will also be a solution of the equations (by making one of the s_j smaller) and will give a larger value of z .

The mechanics of this process involves writing the given information in a matrix. The matrix of the system of equalities including the surplus variables is $(AI_m b)$; the simplex method begins with this matrix above a row beginning with $-c$ followed by zeros. The last element of the last row is the value of the objective function at the current solution. The resulting matrix (the *simplex tableau*) looks like

$$\begin{pmatrix} A & I_m & b \\ -c & 0 & 0 \end{pmatrix}$$

The simplex method involves doing a specific series of row operations that effectively determine a new solution to the constraints that increase the values of the current values of the variables x_i and so increase the value of the objective function. The method ends when there are no negative values left in the bottom row.

`simplex_reduce`

The `simplex_reduce` command performs the simplex method on a linear programming problem of the form, for $b \geq 0$, $Ax \leq b$ with $x \geq 0$.

- `simplex_reduce` takes three arguments:
 - A , b and c from the problem.
- `simplex_reduce(A, b, c)` returns a sequence consisting of:
 - The maximum value of the objective function.
 - The solution of x (augmented since the algorithm works by adding auxiliary variables).
 - The reduced matrix.

Alternatively, you could combine the arguments A, b, c into the matrix that begins the algorithm:

$$\begin{pmatrix} A & I_m & b \\ -c & 0 & 0 \end{pmatrix}$$

which you can get with:

```
B := augment(A, idn(m));
C := border(B, b)
d:=append(-c, 0$(m+1))
D:=augment(C, [d])
and then call simplex_reduce(D).
```

Example.

Find

$$\max(X + 2Y) \text{ where } \begin{cases} (X, Y) \geq 0 \\ -3X + 2Y \leq 3 \\ X + Y \leq 4 \end{cases}$$

Input:

```
simplex_reduce([[ -3, 2], [1, 1]], [3, 4], [1, 2])
```

Output:

$$7, [1, 3, 0, 0], \begin{bmatrix} 0 & 1 & \frac{1}{5} & \frac{3}{5} & 3 \\ 1 & 0 & -\frac{1}{5} & \frac{1}{5} & 1 \\ 0 & 0 & \frac{1}{5} & \frac{7}{5} & 7 \end{bmatrix}$$

This means that the maximum of $X + 2Y$ under these conditions is 7, it is obtained for $X = 1, Y = 3$ because $[1, 3, 0, 0]$ is the augmented solution and the reduced matrix is:

$$\begin{bmatrix} 0 & 1 & \frac{1}{5} & \frac{3}{5} & 3 \\ 1 & 0 & -\frac{1}{5} & \frac{1}{5} & 1 \\ 0 & 0 & \frac{1}{5} & \frac{7}{5} & 7 \end{bmatrix}$$

Notice that the (non-zero) values of the solution have columns of the $m \times m$ identity matrix above them.

Reducing a more complicated case to the simple case

With the former call of `simplex_reduce`, you have to:

- rewrite constraints to the form $x_k \geq 0$,
- remove variables without constraints,
- add variables such that all the constraints have positive components.

For example, find:

$$\min(2x + y - z + 4) \quad \text{where} \quad \begin{cases} x \leq 1 \\ y \geq 2 \\ x + 3y - z = 2 \\ 2x - y + z \leq 8 \\ -x + y \leq 5 \end{cases} \quad (6.11)$$

Let $x = 1 - X$, $y = Y + 2$, $z = 5 - X + 3Y$. The problem is equivalent to finding the minimum of $(-2X + Y - (5 - X + 3Y) + 8)$ where:

$$\begin{cases} X \geq 0 \\ Y \geq 0 \\ 2(1 - X) - (Y + 2) + 5 - X + 3Y \leq 8 \\ -(1 - X) + (Y + 2) \leq 5 \end{cases}$$

or to find the minimum of:

$$(-X - 2Y + 3) \quad \text{where} \quad \begin{cases} X \geq 0 \\ Y \geq 0 \\ -3X + 2Y \leq 3 \\ X + Y \leq 4 \end{cases}$$

i.e. to find the maximum of $-(-X - 2Y + 3) = X + 2Y - 3$ under the same conditions, hence it is the same problem as to find the maximum of $X + 2Y$ seen before. You found that the previous problem had a maximum of 7, hence the result here is $7 - 3 = 4$.

The general case

A linear programming problem may not in general be directly reduced like above to the simple case. The reason is that a starting solution must be found before applying the simplex algorithm.

The standard form of a linear programming problem is similar to the simplest case above, but with $Ax = b$ (instead of $Ax \leq b$) for $b \geq 0$ under the conditions $x \geq 0$. In this case, there is no straightforward solution. So the first problem is to find an x with $x \geq 0$ and $Ax = b$.

Finding a starting solution Let m be the number of rows of A . Add artificial variables s_1, \dots, s_m and maximize $-\sum s_i$ under the conditions $Ax = b, x \geq 0, s \geq 0$ starting with initial value 0 for x variables and $y = b$. If a solution exists and is 0, then the s_i must all be 0 and so x will be solution of $Ax = b, x \geq 0$.

You can solve this with `simplex_reduce` by calling it with a single matrix argument:

$$\begin{pmatrix} A & I_m & b \\ 0 & 1 & 0 \end{pmatrix}$$

For example, to find the minimum of $2x + 3y - z + t$ with $x, y, z, t \geq 0$ and:

$$\begin{aligned} -x - y + t &= 1 \\ y - z + t &= 3 \end{aligned}$$

you would start by finding a solution of

$$\begin{aligned} -x - y + t + s_1 &= 1 \\ y - z + t + s_2 &= 3 \end{aligned}$$

as mentioned above.

Input:

```
simplex_reduce([[ -1, -1, 0, 1, 1, 0, 1], [0, 1, -1, 1, 0, 1, 3], [0, 0, 0, 0, 1, 1, 0]])
```

Output:

$$0, [0, 1, 0, 2, 0, 0], \begin{bmatrix} -\frac{1}{2} & 0 & -\frac{1}{2} & 1 & \frac{1}{2} & \frac{1}{2} & 2 \\ \frac{1}{2} & 1 & -\frac{1}{2} & 0 & -\frac{1}{2} & \frac{1}{2} & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

Since the solution $(x, y, z, t, s_1, s_2) = (0, 1, 0, 2, 0, 0)$ has $s_1 = s_2 = 0$, a non-negative solution of the equalities in x, y, z and t is $(x, y, z, t) = (0, 1, 0, 2)$.

Finding a solution of the LP problem Now you can make a second call to `simplex_reduce` with the reduced matrix (less the columns corresponding to the variables s_1 and s_2) and the usual c . In this case, finding the requested minimum means finding the maximum of $-2x - 3y + z - t$, so $c = (-2, -3, 1, -1)$.

Input:

```
simplex_reduce([[ -1/2, 0, -1/2, 1, 2], [1/2, 1, -1/2, 0, 1], [2, 3, -1, 1, 0]])
```

Output:

$$-5, [0, 1, 0, 2], \begin{bmatrix} -\frac{1}{2} & 0 & -\frac{1}{2} & 1 & 2 \\ \frac{1}{2} & 1 & -\frac{1}{2} & 0 & 1 \\ 1 & 0 & 1 & 0 & -5 \end{bmatrix}$$

This maximum is -5, so the requested minimum is $-(-5) = 5$ at $(0, 1, 0, 2)$ ($x = 0$, $y = 1$, $z = 0$ and $t = 2$).

6.52.2 Solving general linear programming problems: `lpsolve`

The `lpsolve` command can solve linear programming problems, where a multivariate linear function needs to be maximized or minimized subject to linear (in)equality constraints, as well as (mixed) integer programming problems. You can either enter a problem directly (in symbolic or matrix form) or load it from a file in LP or (compressed) MPS format.

Solving an LP problem in symbolic form

To enter a problem symbolically:

- `lpsolve` takes one mandatory argument and three optional arguments:
 - *obj*, a symbolic expression representing the objective function or a path to a file containing the LP problem.
 - Optionally, *constr*, list of linear constraints which may be equalities or inequalities or doubly bounded expressions in form $a \leq f(x) \leq b$ entered as `f(x)=a..b`. (If *obj* is a file name, this option is omitted.)
 - Optionally, *bd*, a sequence of expressions of type `x=a..b`, specifying that the variable x is bounded below by $a \in \mathbb{R} \cup \{-\infty\}$ and above by $b \in \mathbb{R} \cup \{+\infty\}$. If the bounds are stored in a two-column matrix, consider using the conversion routine `box_constraints` (see Section 6.38.9).
 - Optionally, *opts*, a sequence of solver settings in form *option* = *value*, where *option* may be one of:
 - * `lp_assume` (or `assume`), which specifies a global constraint on variables and whose value can be one of:
 - `lp_nonnegative` (all variables are nonnegative)

- `lp_integer` or `integer` (all variables are integers)
 - `lp_binary` (all variables are binary, i.e. 0 or 1)
 - `lp_nonnegint` or `nonnegint` (all variables are nonnegative integers)
- (by default, *unset*.)
- * `lp_integervariables`, whose value should be a list of identifiers or indices of integer variables (empty by default).
 - * `lp_binaryvariables`, whose value should be a list of identifiers or indices of binary variables (empty by default).
 - * `lp_maximize` (or `maximize`), whose value can be `true` or `false` setting the objective direction (by default `false`, meaning that the objective is minimized). You can enter only `maximize`, which is equivalent to `maximize=true`.
 - * `lp_method`, setting the solver type, which can be one of:
 - `exact`
 - `float`
 - `lp_simplex`
 - `lp_interiorpoint`
 (by default, `lp_method=lp_simplex`).
 - * `lp_depthlimit`, whose value can be a positive integer, which sets the maximum depth of the branch-and-bound tree (by default, *unlimited*).
 - * `lp_nodelimit`, whose value can be a positive integer, which sets the maximum number of nodes in the branch-and-bound tree (by default, *unlimited*).
 - * `lp_iterationlimit`, whose value can be a positive integer setting the maximum iterations of the simplex algorithm (by default, *unlimited*).
If the maximum number of iterations is reached, the current feasible solution (not necessarily an optimal one) is returned.
 - * `lp_timelimit`, whose value can be a positive real number, setting the maximum solving time in milliseconds (by default, *unlimited*).
 - * `acyclic`, whose value can be either `true` or `false`, enabling/disabling the Bland rule safeguarding (by default, `true`).
 - * `lp_maxcuts`, whose value can be a nonnegative integer setting the maximum GMI cuts per node (by default 5).
 - * `lp_gaptolerance`, whose value can be a positive number, setting the relative integrality gap threshold (by default, 0).
 - * `lp_presolve`, whose value can be either `true` or `false`, enabling/disabling the preprocessing step (by default, `true`).
 - * `lp_heuristic`, whose value can be either `true` or `false`, enabling/disabling the rounding heuristic (by default, `true`).
 - * `lp_nodeselect`, which sets the branching node selection strategy and whose value can be one of:

- `lp_depthfirst`
- `lp_breadthfirst`
- `lp_bestlocalbound`
- `lp_hybrid`
- `lp_bestprojection`
- (by default, `lp_nodeselect=lp_bestlocalbound`).
- * `lp_vartselect`, which sets the branching variable selection strategy, whose value can be one of
 - `lp_firstfractional`
 - `lp_lastfractional`
 - `lp_mostfractional`
 - `lp_pseudocost`
 (by default, `lp_vartselect=lp_pseudocost`).
- * `lp_verbose`, whose value can be `true` or `false` (by default `false`). You can enter only `lp_verbose`, which is equivalent to `lp_verbose=true`.
- `lpsolve(obj, <constr, bd, opts>)` returns a list `[optimum, soln]`, where *optimum* is the minimum/maximum value of the objective function and *soln* is the list of coordinates corresponding to the point at which the optimal value is attained, i.e. the optimal solution. If there is no feasible solution, an empty list is returned. When the objective function is unbounded, *optimum* is returned as `+infinity` (for maximization problems) or `-infinity` (for minimization problems). If an error is experienced while solving (terminating the process), then `undef` is returned.

The given objective function is minimized by default. To maximize it, include the option `lp_maximize=true` or `lp_maximize` or simply `maximize`. Also note that all variables are, unless specified otherwise, assumed to be continuous and unrestricted in sign.

Examples.

- Solve the problem specified in (6.11):

Input:

```
constr:=[x<=1,y>=2,x+3y-z=2,3x-y+z<=8,-x+y<=5];
lpsolve(2x+y-z+4,constr)
```

Output:

```
[-4, [x = 0, y = 5, z = 13]]
```

Therefore, the minimum value of $f(x, y, z) = 2x + y - z + 4$ is equal to -4 under the given constraints. The optimal value is attained at point $(x, y, z) = (0, 5, 13)$.

- Constraints may also take the form `expr=a..b` for bounded linear expressions. For example, minimize $x + 2y + 3z$ subject to $1 \leq x + y \leq 5$ and $2 \leq y + z + 1 \leq 4$, where $x, y \geq 0$.

Input:

```
lpsolve(x+2y+3z,[x+y=1..5,y+z=1..2,x>=0,y>=0])
```

Output:

$$[-2, [x = 0, y = 5, z = -4]]$$

- Use the `assume=lp_nonnegative` option to specify that all variables are nonnegative. It is easier than entering the nonnegativity constraints explicitly. For example, minimize $-x - y$ subject to $y \leq 3x + \frac{1}{2}$ and $y \leq -5x + 2$, where $x, y \geq 0$.

Input:

```
lpsolve(-x-y,[y<=3x+1/2,y<=-5x+2],assume=lp_nonnegative)
```

Output:

$$\left[-\frac{5}{4}, \left[x = \frac{3}{16}, y = \frac{17}{16}\right]\right]$$

- Bounds can be added separately for some variables. They should be entered after the constraints. For example, minimize $-6x + 4y + z$ subject to

$$\begin{aligned} 5x - 10y &\leq 20, \\ 2z - 3y &= 6, \\ -x + 3y &\leq 3, \end{aligned}$$

where $1 \leq x \leq 20$ and $y \geq 0$.

Input:

```
constr:=[5x-10y<=20,2z-3y=6,-x+3y<=3];
lpsolve(-6x+4y+z,constr,x=1..20,y=0..inf)
```

Output:

$$\left[-\frac{133}{2}, \left[x = 18, y = 7, z = \frac{27}{2}\right]\right]$$

Solving an LP problem in matrix form

To enter a problem in matrix form:

- `lpsolve` takes
 - *obj*, a vector of coefficients representing the objective function.
 - *constr*, a list $[A, b, A_{\text{eq}}, b_{\text{eq}}]$ such that objective function $\text{obj}^T \cdot x$ is to be minimized/maximized subject to constraints $Ax \leq b$ and $A_{\text{eq}}x = b_{\text{eq}}$.
If the problem does not contain equality constraints, A_{eq} and b_{eq} may be omitted. For a problem that does not contain inequality constraints, empty lists must be entered in place of A and b .
 - *bd*, a list of two vectors $[b_l, b_u]$ of the same length as c such that $b_l \leq x \leq b_u$. These vectors may contain `+infinity` or `-infinity`.

– *opts*, as before.

- `lpsolve(obj,⟨constr,bd,opts⟩)` returns a list $[optimum, soln]$ as before. *optimum* is the minimum/maximum value of the objective function and *soln* is the list of coordinates corresponding to the point at which the optimal value is attained, i.e. the optimal solution. If there is no feasible solution, an empty list is returned. When the objective function is unbounded, *optimum* is returned as `+infinity` (for maximization problems) or `-infinity` (for minimization problems). If an error is experienced while solving (terminating the process), `undef` is returned.

Examples.

- *Input:*

```
c:=[-2,1];A:=[[-1,1],[1,1],[-1,0],[0,-1]];b:=[3,5,0,0];
lpsolve(c,[A,b])
```

Output:

$$[-10, [5, 0]]$$

- *Input:*

```
c:=[-2,5,-3];b1:=[2,3,1];bu:=[6,10,7/2];
lpsolve(c,[],[b1,bu])
```

Output:

$$\left[-\frac{15}{2}, \left[6, 3, \frac{7}{2}\right]\right]$$

Input:

```
c:=[4,5];Aeq:=[[-1,3/2],[-3,2]];beq:=[2,3];
lpsolve(c,[],[],Aeq,beq)
```

Output:

$$\left[\frac{26}{5}, \left[-\frac{1}{5}, \frac{6}{5}\right]\right]$$

Simplex method implementation. Only the two-phase primal simplex method is implemented for `lpsolve` and it uses the upper-bounding technique. Basic columns are not kept in the simplex tableau, which therefore contains only the columns of non-basic variables. To prevent cycling, an adaptation of Bland's rule is used. A preprocessing routine, helping to reduce the size of the problem, is available and enabled by default (you can disable it by typing `lp_presolve=false`). All computation is done by using arbitrary-precision integer arithmetic, which is always exact but slower than the floating-point arithmetic. Note that all problem data should be rational.

Cycling in simplex algorithm may happen, although it is rare in practice. Bland rule safeguarding is used by default in order to prevent cycling. However, Bland's pivoting rule is known to converge slowly; therefore it may slow down the computation in problems which would otherwise not cycle. To disable the safeguarding, use the option `acyclic=false`.

Solving MIP (Mixed Integer Programming) problems

The `lpsolve` command allows restricting (some) variables to integer values. Such problems, called (*mixed*) *integer programming problems*, are solved by applying the branch-and-bound method.

To solve pure integer programming problems, in which all variables are integers, use the option `assume=integer` or `assume=lp_integer`.

Example.

Input:

```
lpsolve(-5x-7y,[7x+y<=35,-x+3y<=6],assume=integer)
```

Output:

```
[-41,[x = 4,y = 3]]
```

Use the option `assume=lp_binary` to specify that all variables are binary, i.e. the only allowed values are 0 and 1. These usually represent `false` and `true`, respectively, giving the variable a certain meaning in a logical context.

Example.

Input:

```
lpsolve(8x1+11x2+6x3+4x4,[5x1+7x2+4x3+3x4<=14],
      assume=lp_binary,maximize)
```

Output:

```
[21,[x1 = 0,x2 = 1,x3 = 1,x4 = 1]]
```

To solve mixed integer problems, where some variables are integers and some are continuous, use the option keywords `lp_integervariables` to specify integer variables and/or `lp_binaryvariables` to specify binary variables.

Input:

```
lpsolve(x+3y+3z,[x+3y+2z<=7,2x+2y+z<=11],
      assume=lp_nonnegative,lp_maximize,lp_integervariables=[x,z])
```

Output:

```
[10,[x=1,y=0,z=3]]
```

Use the `assume=lp_nonnegint` or `assume=nonnegint` option to get non-negative integer values.

Example.

Input:

```
lpsolve(2x+5y,[3x-y=1,x-y<=5],assume=nonnegint)
```

Output:

```
[12,[x = 1,y = 2]]
```

When specifying MIP problems in matrix form, the lists corresponding to the options `lp_integervariables` and `lp_binaryvariables` should contain indices of the variables, as in the following example.

Example.

Input:

```
c:=[2,-3,-5];A:[[-5,4,-5],[2,5,7],[2,-3,4]];b:=[3,1,-2];
lpsolve(c,[A,b],lp_integervariables=[0,2])
```

Output:

$$\left[\frac{19}{4}, \left[1, \frac{3}{4}, -1 \right] \right]$$

You can also specify a range of indices instead of a list when there is too much variables. Example: `lp_binaryvariables=0..99` means that all variables x_i such that $0 \leq i \leq 99$ are binary.

Branch-and-bound implementation. The branch-and-bound algorithm generates a binary tree of subproblems, called *nodes*, by branching on integer variables with fractional values. Leaf nodes of the tree, called *active nodes*, are stored in a list. In each iteration of the algorithm, an active node is selected, branched on a particular variable into two new nodes, and subsequently removed from the list. A node in which no branching is possible represents a feasible solution. The corresponding objective value is used to fathom nodes which cannot possibly lead to a better solution. The algorithm terminates when there is no space left for improvement.

If presolving is enabled, then basic preprocessing is done at each node of the tree, except when `lp_presolve=root` is set, in which case only the root node is processed. Additionally, after a non-integer-feasible solution with better objective value than the current incumbent is obtained by solving the linear relaxation, a rounding heuristic is applied in attempt to achieve integral feasibility. The heuristic is enabled by default; you can disable it by setting `lp_heuristic` to `false`.

Node-selection strategies. A node-selection strategy can be chosen by using the `lp_nodeselect` option. Possible values are:

- `lp_bestlocalbound`, which chooses an active node having the best bound for the objective value,
- `lp_depthfirst`, which chooses the newest active node,
- `lp_hybrid`, which combines the above two strategies,
- `lp_breadthfirst`, which chooses the oldest active node,
- `lp_bestprojection`, which chooses a node with the best simple projection.

By default, the `lp_bestlocalbound` strategy is used. The `lp_hybrid` strategy works as follows: until an incumbent solution is found, the solver uses the `lp_depthfirst` strategy, “diving” into the tree as an incumbent solution is more likely to be located deeply. When an incumbent is found, the solver switches to the `lp_bestlocalbound` strategy in attempt to close the integrality gap as quickly as possible.

Variable-selection strategies. A variable-selection strategy can be chosen by using the `lp_vartselect` option. Possible values are:

- `lp_firstfractional`, which chooses the first fractional variable,
- `lp_lastfractional`, which chooses the last fractional variable,
- `lp_mostfractional`, which chooses a variable with fractional part closest to 0.5,
- `lp_pseudocost`, which chooses the variable which had the greatest impact on the objective value in previous branchings.

By default, the `lp_pseudocost` strategy is used. However, since a pseudocost-based choice cannot be made until all integer variables have been branched upon at least one time in each direction, the `lp_mostfractional` strategy is used until that condition is fulfilled.

Using an appropriate combination of node/variable selection strategies may significantly reduce the number of subproblems needed to be examined when solving a particular MIP problem, as illustrated in the following example.

Example.

Minimize $z = \mathbf{c} \cdot \mathbf{x}$ subject to $\mathbf{A} \mathbf{x} = \mathbf{b}$, where $\mathbf{x} \in \mathbb{Z}_+^8$ and

$$\mathbf{A} = \begin{bmatrix} 22 & 13 & 26 & 33 & 21 & 3 & 14 & 26 \\ 39 & 16 & 22 & 28 & 26 & 30 & 23 & 24 \\ 18 & 14 & 29 & 27 & 30 & 38 & 26 & 26 \\ 41 & 26 & 28 & 36 & 18 & 38 & 16 & 26 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} 7872 \\ 10466 \\ 11322 \\ 12058 \end{bmatrix}, \quad \mathbf{c} = \begin{bmatrix} 2 \\ 10 \\ 13 \\ 17 \\ 7 \\ 5 \\ 7 \\ 3 \end{bmatrix}.$$

The optimal solution is $z^* = 1854$, $\mathbf{x}^* = (24, 15, 19, 11, 3, 99, 4, 226)$. In the following table, different strategies are compared according to the number of examined nodes and total solving time (in seconds).

Node selection	Variable selection	Nodes examined	Time
Best local bound	Last fractional	13102	4.8
Best projection	Most fractional	26238	11.7
Hybrid	First fractional	55046	19.5
Depth-first	Pseudocost	131466	36.2

Note that the above comparison is problem-specific; it does not mean that `lp_bestlocalbound+lp_lastfractional` is always the best strategy. Usually, one has to experiment with different combinations to find which one is optimal for the given problem. However, the strategies which use larger amounts of information generally perform better. Therefore, it is reasonable to assume that `lp_bestlocalbound` will be more appropriate than `lp_breadthfirst`, for instance.

Cutting planes. Gomory mixed integer (GMI) cuts are generated at every node of the branch-and-bound tree to improve the objective value bound. After solving the relaxed subproblem using the simplex method, GMI cuts are added to the subproblem which is subsequently reoptimized. This procedure is repeated until no useful cuts can be generated or until `lp_maxcuts` limit is reached.

Simplex reoptimizations are fast because they start with the last feasible basis; however, applying cuts makes the simplex tableau larger, which may slow the computation down. To limit the number of GMI cuts that is allowed be appended to a subproblem, you can use `lp_maxcuts` option, setting it either to zero (which disables the cut generation altogether) or to some positive integer. You may set it to `+infinity` as well, thus allowing any number of cuts to be applied to a node. By default, `lp_maxcuts` equals to 5.

Stopping criteria. There are several ways to force the branch-and-bound algorithm to stop prematurely when the execution takes too much time. You can set `lp_timelimit` to an integer which defines the maximum number of milliseconds allowed to find an optimal solution. Other ways are to set `lp_nodelimit` or `lp_depthlimit` to limit the number of nodes generated in the branch-and-bound tree or its depth, respectively. Finally, you can set `lp_gaptolerance` to some positive value, say $t > 0$, which terminates the algorithm after finding an incumbent solution and proving that the corresponding objective value differs from the optimum value for less than $t \cdot 100\%$. This is done by monitoring the size of the integrality gap, i.e. the difference between the current incumbent objective value and the best objective value bound among active nodes.

If the branch-and-bound algorithm terminates prematurely, a warning message indicating the cause is displayed. The incumbent solution, if any, is returned as the result, else the problem is declared to be infeasible.

Displaying detailed output. Typing `lp_verbose=true` or `lp_verbose` when specifying options for `lpsolve` causes detailed messages to be printed during and after solving a LP problem. During the simplex algorithm, a status report in form

$$n \text{ obj: } z$$

is printed every two seconds, where n is the number of simplex iterations and z is the current objective value. If the line is prefixed with the asterisk character (*), it means that the solver is optimizing the given objective (Phase 2); otherwise, the solver is searching for a feasible basis (Phase 1), in which case z is a relative value in percentages (when it reaches zero, Phase 2 is initiated). Note that values of z reported in Phase 2 may not correspond to the actual values if presolving is enabled.

If the problem contains integrality constraints, then the simplex algorithm messages are not printed. Instead, during the branch-and-bound algorithm, a status report in form

```
n:  m nodes active, bound:  b< gap:  g>
```

is displayed every 5 seconds, where n is the number of already examined subproblems, b is the lower resp. upper bound (for minimization resp. maximization), and g is the relative integrality gap. Also, a message is printed every time the incumbent solution is found or updated, as well as when the solver switches to pseudocost-based branching. After the algorithm is finished, a summary is displayed containing the total number of examined subproblems, the number of most nodes being active at the same time, and the number of applied GMI cuts along with the respective average objective value improvement.

In the following example, we solve the MIP given above. The solver shows all progress and summary messages.

Example.

Input:

```
A:=[ [22,13,26,33,21,3,14,26], [39,16,22,28,26,30,23,24],
      [18,14,29,27,30,38,26,26], [41,26,28,36,18,38,16,26] ];;
b:=[7872,10466,11322,12058];;c:=[2,10,13,17,7,5,7,3];;
lpsolve(c, [], [], A, b), assume=nonnegint, lp_verbose)
```

Output:

```
Constraint matrix has 4 rows, 9 columns, and 36 nonzeros
Variables: 0 continuous, 8 integer (0 binary)
Constraints: 4 equalities, 0 inequalities
Optimizing...
Starting branch & bound...
11310: 147 nodes active, bound: 1793.43
12709: Incumbent solution found: 1854
20646: 931 nodes active, bound: 1841.81, gap: 0.657343%
23836: Tree is empty
Summary:
* 23836 subproblem(s) examined
* max. tree size: 1256 nodes
* 13 GMI cut(s) applied (average improvement: 2.34241e-05%)
[1854, [24, 15, 19, 11, 3, 99, 4, 226]]
```

Solving problems in floating-point arithmetic

The `lpsolve` command provides, in addition to its own exact solver implementing the primal simplex method with the upper-bounding technique, an interface to GLPK (GNU Linear Programming Kit) library which contains sophisticated LP/MIP solvers in floating-point arithmetic, designed to be very fast and to handle large problems. Choosing between the available solvers is done by setting `lp_method` option.

By default, `lp_method` is set to `lp_simplex`, which solves the problem by using the native solver, but only if all problem coefficients are exact. If at least one of them is approximative (a floating-point number), then the GLPK solver is used instead.

Setting `lp_method` to `exact` forces the solver to perform exact computation even when some coefficients are inexact (they are converted to rational equivalents before applying the simplex method). In this case the native solver is used.

Setting `lp_method` to `float` forces `lpsolve` to use the GLPK simplex solver. If a (mixed) integer problem is given, then the branch-and-cut algorithm in GLPK is used. The parameters can be controlled by setting the `lp_timelimit`, `lp_iterationlimit`, `lp_gaptolerance`, `lp_maxcuts`, `lp_heuristic`, `lp_verbose`, `lp_nodeselect`, and `lp_varselect` options. If `lp_varselect` is not set, then the Driebeek–Tomlin heuristic is used, and if `lp_nodeselect` is not set, then the best-local-bound selection method is used. If `lp_maxcuts` is greater than zero, then GMI/MIR cut generation is enabled, else it is disabled. If the problem contains binary variables, then cover/clique cut generation is enabled, else it is disabled. If `lp_heuristic=false`, then the simple rounding heuristic in GLPK is disabled (by default it is enabled). Finally, `lp_verbose=true` enables GLPK messages, which are useful for monitoring solver's progress.

Setting `lp_method` to `lp_interiorpoint` uses the primal-dual interior-point algorithm in GLPK. The only optional argument that affects this kind of solver is `lp_verbose`. The interior-point algorithm is faster than the simplex method for large and sparse LP problems. Note, however, that it does not support integrality constraints.

For example, we solve the following LP problem using the default settings.

$$\text{Minimize } 1.06x_1 + 0.56x_2 + 3.0x_3$$

subject to

$$\begin{aligned} 1.06x_1 + 0.015x_3 &\geq 729824.87 \\ 0.56x_2 + 0.649x_3 &\geq 1522188.03 \\ x_3 &\geq 1680.05 \end{aligned}$$

Input:

```
lpsolve(1.06x1+0.56x2+3x3,
  [1.06x1+0.015x3>=729824.87,0.56x2+0.649x3>=1522188.03,x3>=1680.05])
```

Output:

```
[2255937.4968, [x1 = 688490.254009, x2 = 2716245.85277, x3 = 1680.05]]
```

If the requirement $x_k \in \mathbb{Z}$ for $k = 1, 2$, the following result is obtained from the MIP solver in GLPK.

Input:

```
lpsolve(1.06x1+0.56x2+3x3,
  [1.06x1+0.015x3>=729824.87,0.56x2+0.649x3>=1522188.03,x3>=1680.05],
  lp_integervariables=[0,1])
```

Output:

$[2255938.37, [x_1 = 688491, x_2 = 2716246, x_3 = 1680.05]]$

The solution of the original problem can also be obtained with the interior-point solver by using the `lp_method=lp_interiorpoint` option.

Input:

```
lpsolve(1.06x1+0.56x2+3x3,
        [1.06x1+0.015x3>=729824.87,0.56x2+0.649x3>=1522188.03,x3>=1680.05],
        lp_method=lp_interiorpoint)
```

Output:

$[2255937.50731, [x_1 = 688490.256652, x_2 = 2716245.85608, x_3 = 1680.05195065]]$

Loading a problem from a file

Linear (integer) programming problems can be loaded from MPS¹ or CPLEX LP² format files. The file name should be a string passed as the *obj* parameter. If the file name has extension `.lp`, then CPLEX LP format is assumed. If the extension is `.mps` resp. `.gz`, then MPS resp. gzipped MPS format is assumed.

For example, assume that the file `somefile.lp` is stored in the working directory and that it contains the following lines of text:

```
Maximize
  obj: 2x1 + 4x2 + 3x3
Subject To
  c1: 3x1 + 5x2 + 2x3 <= 60
  c2: 2x1 + x2 + 2x3 <= 40
  c3: x1 + 3x2 + 2x3 <= 80
General
  x1 x3
End
```

To find an optimal solution to the linear program specified in this file, enter:

Input:

```
lpsolve("somefile.lp")
```

Output:

$[71.8, [x_1 = 0, x_2 = 5.2, x_3 = 17]]$

You can provide additional variable bounds, assumptions, and options alongside the file name, as in the examples below. Note that the original constraints (those which are read from file) cannot be removed.

Input:

```
lpsolve("somefile.lp",x2=5.4..inf,lp_method=exact)
```

¹<http://web.mit.edu/lpsolve/doc/mps-format.htm>

²<http://web.mit.edu/lpsolve/doc/CPLEX-format.htm>

Output:

$$\left[\frac{352}{5}, \left[x_1 = 0, x_2 = \frac{28}{5}, x_3 = 16 \right] \right]$$

Next we force all variables to integral values.

Input:

```
lpsolve("somefile.lp",x2=5.4..inf,assume=integer)
```

Output:

$$[69.0, [x_1 = 0, x_2 = 6, x_3 = 15]]$$

It is advisable to use only (capital) letters, digits and underscores when naming variables in an LP file, although the corresponding format allows many more characters. That is because these names are converted to **giac** identifiers during the loading process.

Note that the coefficients of a problem loaded from file are always floating-point values. Therefore, to solve the problem with the native solver, you have to use the option **lp_method=exact** (see Section 6.52.2).

6.52.3 Solving the transportation problems: **tpsolve**

The objective of a transportation problem is to minimize the cost of distributing a product from m sources to n destinations. It is determined by three parameters:

- $\mathbf{s} = (s_1, s_2, \dots, s_m)$, the supply vector, where $s_k \in \mathbb{Z}^+$ is the maximum number of units that can be delivered from the k th source for $k = 1, 2, \dots, m$,
- $\mathbf{d} = (d_1, d_2, \dots, d_n)$, the demand vector $\mathbf{d} = (d_1, d_2, \dots, d_n)$, where $d_k \in \mathbb{Z}^+$ is the minimum number of units required by the k th destination for $k = 1, 2, \dots, n$
- $\mathbf{C} = [c_{ij}]_{m \times n}$, the cost matrix, where $c_{ij} \in \mathbb{R}^+$ is the cost of transporting one unit of product from the i th source to the j th destination, for $i = 1, 2, \dots, m$ and $j = 1, 2, \dots, n$.

An optimal solution is represented by the matrix $\mathbf{X}^* = (x_{ij}^*)$, where x_{ij}^* is number of units that must be transported from the i th source to the j th destination for $i = 1, 2, \dots, m$ and $j = 1, 2, \dots, n$.

The **tpsolve** command solves the transportation problem.

- **tpsolve** takes three arguments:
 - s , the supply vector.
 - d , the demand vector.
 - C , the cost matrix.
- **tpsolve**(s, d, C) returns a sequence c, X^* , where X^* is the optimal solution and $c = \sum_{i=1}^m \sum_{j=1}^n c_{ij} x_{ij}^*$ is the total (minimal) cost of transportation.

Example.*Input:*

```
s:=[12,17,11];d:=[10,10,10,10];
C:=[[50,75,30,45],[65,80,40,60],[40,70,50,55]];
tpsolve(s,d,C)
```

Output:

$$2020, \begin{bmatrix} 0 & 0 & 2 & 10 \\ 0 & 9 & 8 & 0 \\ 10 & 1 & 0 & 0 \end{bmatrix}$$

If the total supply and total demand are equal, i.e. if $\sum_{i=1}^m s_i = \sum_{j=1}^n d_j$ holds, the transportation problem is said to be *closed* or *balanced*, otherwise it is said to be *unbalanced*. The excess supply/demand is covered by adding a dummy demand/supply point with zero cost of “transportation” from/to that point. The `tpsolve` command handles such cases automatically.

Example.*Input:*

```
s:=[7,10,8,8,9,6];d:=[9,6,12,8,10];
C:=[[36,40,32,43,29],[28,27,29,40,38],[34,35,41,29,31],
[41,42,35,27,36],[25,28,40,34,38],[31,30,43,38,40]];
tpsolve(s,d,C)
```

Output:

$$1275, \begin{bmatrix} 0 & 0 & 2 & 0 & 5 \\ 0 & 0 & 10 & 0 & 0 \\ 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 0 & 8 & 0 \\ 9 & 0 & 0 & 0 & 0 \\ 0 & 6 & 0 & 0 & 0 \end{bmatrix}$$

Sometimes it is desirable to forbid transportation on certain routes. That is usually achieved by setting a very high cost, represented by a symbol (usually M), to these routes. Hence all instances of a symbol in the cost matrix are automatically replaced by a practically infinite number (precisely $(mn+1)/\varepsilon$, where ε is returned by `epsilon`), which forces the algorithm to avoid the associated routes.

Example.*Input:*

```
s:=[95,70,165,165];d:=[195,150,30,45,75];
C:=[[15,M,45,M,0],[12,40,M,M,0],[0,15,25,25,0],[M,0,M,12,0]]
tpsolve(s,d,C)
```

Output:

$$2820, \begin{bmatrix} 20 & 0 & 0 & 0 & 75 \\ 70 & 0 & 0 & 0 & 0 \\ 105 & 0 & 30 & 30 & 0 \\ 0 & 150 & 0 & 15 & 0 \end{bmatrix}$$

6.53 Nonlinear optimization

6.53.1 Global extrema: minimize maximize

minimize attempts to find the smallest value of a differentiable expression defined on a compact domain, using analytical methods.

- **minimize** takes two mandatory arguments and two optional arguments:

- *obj*, a univariate or multivariate expression
- Optionally, *constr*, list of constraints given by equalities, inequalities, and/or expressions assumed to be equal to 0. If there is only one constraint, it doesn't have to be a list.
- *vars*, list of variables. If there is only one variable, it doesn't have to be a list. A variable can also include bounds, as in $x = a..b$.
- Optionally, *location*, a keyword which may be **coordinates**, **locus** or **point**.

- **minimize**(*obj*⟨, *constr*⟩, *vars*⟨, *location*⟩) returns the minimum value of *obj* on the domain specified by constraints and/or bounding variables, or (if *location* is specified) a list consisting of the minimum value and the list of points where the minimum is achieved.

The domain must be closed and bounded (i.e. compact) and *obj* must be continuous and differentiable except in a finite number of points; otherwise the final result may be incorrect or meaningless. If the minimal value could not be obtained, then **undef** is returned.

The **maximize** command takes the same parameters as **minimize**, but returns the global maximum of *obj* on the specified domain.

Examples.

- *Input:*

```
minimize(sin(x), [x=0..4])
```

Output:

$\sin(4)$

- *Input:*

```
minimize(asin(x), x=-1..1)
```

Output:

$-\frac{\pi}{2}$

- *Input:*

```
minimize(x^4-x^2, x=-3..3, locus)
```

Output:

$$\left[-\frac{1}{4}, \left[\frac{\sqrt{2}}{2}, -\frac{\sqrt{2}}{2} \right] \right]$$

• *Input:*

```
minimize(x-abs(x),x=-1..1)
```

Output:

$$-2$$

• *Input:*

```
minimize(when(x==0,0,exp(-1/x^2)),x=-1..1)
```

Output:

$$0$$

• *Input:*

```
minimize(sin(x)+cos(x),x=0..20,coordinates)
```

Output:

$$\left[-\sqrt{2}, \left[\frac{5}{4}\pi, \frac{13}{4}\pi, \frac{21}{4}\pi \right] \right]$$

• *Input:*

```
minimize(x^2-3x+y^2+3y+3,[x=2..4,y=-4..-2],point)
```

Output:

$$[-1, \begin{bmatrix} 2 & -2 \end{bmatrix}]$$

• *Input:*

```
obj:=sqrt(x^2+y^2)-z;
constr:=[x^2+y^2<=16,x+y+z=10];
minimize(obj,constr,[x,y,z])
```

Output:

$$-4\sqrt{2} - 6$$

• *Input:*

```
minimize(x^2*(y+1)-2y,[y<=2,sqrt(1+x^2)<=y],[x,y])
```

Output:

$$-4$$

• *Input:*


```
maximize(cos(x),x=1..3)
```

Output:

$$\cos(1)$$

• *Input:*

```
obj:=piecewise(x<=-2,x+6,x<=1,x^2,3/2-x/2);
maximize(obj,x=-3..2)
```

Output:

$$4$$

• *Input:*

```
maximize(x*y*z,x^2+2*y^2+3*z^2<=1,[x,y,z])
```

Output:

$$\frac{\sqrt{2}}{18}$$

• *Input:*

```
maximize(x*y,[x+y^2<=2,x>=0,y>=0],[x,y],locus)
```

Output:

$$\left[\frac{4}{9}\sqrt{6}, \left[\frac{4}{3} \quad \frac{\sqrt{6}}{3} \right] \right]$$

• *Input:*

```
maximize(y^2-x^2*y,y<=x,[x=0..2,y=0..2])
```

Output:

$$\frac{4}{27}$$

• *Input:*

```
assume(a>0);
maximize(x^2*y^2*z^2,x^2+y^2+z^2=a^2,[x,y,z])
```

Output:

$$\frac{a^6}{27}$$

6.53.2 Local extrema: extrema

extrema attempts to find local extrema of a univariate/multivariate differentiable expression under equality constraints, using analytical methods.

- **extrema** which takes two mandatory arguments and two optional arguments:

- *expr*, a differentiable expression.
- Optionally, *constr*, a list of equality constraints, where each constraint is an equality or an expression assumed to be equal to zero, and the number of constraints must be strictly less than the number of variables. If there is only one constraint, it doesn't have to be a list.

Additionally, the Jacobian matrix of the constraints must be full rank (i.e., denoting the k th constraint by $g_k(x_1, x_2, \dots, x_n) = 0$ for $k = 1, 2, \dots, m$ and letting $\mathbf{g} = (g_1, g_2, \dots, g_m)$, the Jacobian matrix of \mathbf{g} must be equal to m).

- *vars*, a list of variables. A variable can be specified with bounds, $x = a..b$, where a or b is allowed to be **-infinity** or **infinity**.

If there is only one variable, it doesn't have to be a list.

The parameter **vars** can also be entered as a list of values of the variables; e.g. $[x_1 = a_1, x_2 = a_2, \dots, x_n = a_n]$, in which case the critical point close to $a = (a_1, a_2, \dots, a_n)$ is computed numerically by , applying an iterative method with initial point a .

- Optionally, *opt*, which can be **order=n** to make n the upper bound for the order of derivatives examined in the process (so if $n = 1$ only critical points are found, by default **order=5**) or **lagrange** to specify the method of Lagrange multipliers.

- **extrema**(*expr* < *constr* >, *vars* < *opt* >) returns a list $[min, max]$, where *min* is the list of local minima and *max* is the list of local maxima of *expr*.

Saddle and unclassified points are reported in the message area. Also, information about possible (non)strict extrema is printed out. If **lagrange** is passed as an optional last argument, the method of Lagrange multipliers is used. Else, the problem is reduced to an unconstrained one by applying implicit differentiation.

If critical points are left unclassified you might consider repeating the process with larger value of n , although the success is not guaranteed.

Examples.

- *Input:*

```
extrema(-2*cos(x)-cos(x)^2,x)
```

Output:

$$\begin{bmatrix} 0 \\ \pi \end{bmatrix}$$

- *Input:*

```
extrema(x/2-2*sin(x/2), x=-12..12)
```

Output:

$$\begin{bmatrix} -\frac{10}{3}\pi & \frac{2}{3}\pi \\ -\frac{2}{3}\pi & \frac{10}{3}\pi \end{bmatrix}$$

- *Input:*

```
assume(a>0); extrema(x^2+a*x, x)
```

Output:

$$\left[\left[-\frac{1}{2}a \right], [] \right]$$

- *Input:*

```
extrema(exp(x^2-2x)*ln(x)*ln(1-x), x=0..5)
```

Output:

$$[[], [0.277769149124]]$$

- *Input:*

```
extrema(x^3-2x*y+3y^4, [x,y])
```

Output:

$$\left[\left[\frac{12^{\frac{1}{5}}}{3}, \frac{(12^{\frac{1}{5}})^2}{6} \right], [] \right]$$

- *Input:*

```
assume(a>0); extrema(x/a^2+a*y^2, x+y=a, [x,y])
```

Output:

$$\left[\left[\frac{2a^4-1}{2a^3}, \frac{1}{2a^3} \right], [] \right]$$

- *Input:*

```
extrema(x^2+y^2, x*y=1, [x=0..inf, y=0..inf])
```

Output:

$$[[1 \quad 1], []]$$

- *Input:*

```
extrema(x*y*z, x+y+z=1, [x,y,z], order=1)
```

Output:

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \end{bmatrix}$$

6.53.3 Continuous derivative-free optimization: `nlp_solve`

`nlp_solve` computes the optimum of a continuous, not necessarily differentiable nonlinear (multivariate) objective function, subject to a set of nonlinear equality and/or inequality constraints, using the COBYLA algorithm.

- `nlp_solve` takes the following arguments:

- *obj*, an expression to optimize.
- Optionally, *constr*, a list of equality and inequality constraints. Double-bounded constraints in form $a \leq f(x) \leq b$ can be entered as `f(x)=a..b` instead of two inequalities.
- Optionally, *bd*, a sequence of variable boundaries `x=a..b` where $a \in \mathbb{R} \cup \{-\infty\}$ and $b \in \mathbb{R} \cup \{+\infty\}$, $b \geq a$. If several variables have the same upper and lower bounds, one can enter `[x,y,...]=a..b`. If the bounds are stored in a two-column matrix, consider using the conversion routine `box_constraints` (see Section 6.38.9).
- Optionally, *opt*, a sequence of options in which each option may be one of:
 - * `maximize=bool` or `nlp_maximize=bool`, where *bool* can be `true` or `false`. Just `maximize` or `nlp_maximize` is equivalent to `maximize=true`. (By default, `maximize=false`, i.e. the objective is minimized.)
 - * `nlp_initialpoint=pt`, where *pt* is a starting point for the solver, given in the form $[x = x_0, y = y_0, \dots]$. *pt* does not have to be feasible, but it must not be zero. If none is given or the given point is not feasible, then it is driven into the feasible region before optimizing the objective.
 - * `nlp_samples=n` for a nonnegative integer *n*. If $n > 0$, then *n* points are sampled at random in the feasible region. At most five points with the smallest/largest value of *obj* (in case of minimization/maximization) are chosen for the multistart heuristic which enables `nlp_solve` to reach global optimum in most cases. If $n = 0$, then the randomized standard heuristic is used as a single starting point.³ Note that if `nlp_initialpoint` is set, these heuristics are not used. (By default, $n = 500$.)
 - * `nlp_presolve=bool`, where *bool* can be `true` or `false`. This option enables/disables the preprocessing step which attempts to reduce the number of decision variables by solving subsets of equality constraints which are all linear in certain subsets of variables, substituting the results, and repeating the process until no further reduction can be made. (In particular, fixed variables are removed.) After solving the presolved problem, the removed variables are substituted back to the solution. (By default, `nlp_presolve=true`.)

³See J. W. Chinneck, *Feasibility and Infeasibility in Optimization*, Springer, 2008.

- * `nlp_integervariables=lst`, where *lst* is a list of problem variables that are supposed to take integral values. (By default, there are no integral variables.)
 - * `nlp_binaryvariables=lst`, where *lst* is a list of problem variables that are supposed to take binary (0-1) values. (By default, there are no binary variables.)
 - * `nlp_iterationlimit=n` for a positive integer *n*. This sets the maximum number of iterations allowed for the solver. (By default, $n = 10^5$.)
 - * `nlp_precision= ϵ` for some $\epsilon > 0$. This sets the default precision for the optimizer. (By default, `nlp_precision` is equal to the value of `epsilon()`, which can be set in the session settings in Xcas.)
 - * `nlp_feasibilitytolerance= ϵ` for some $\epsilon > 0$. This sets the feasibility tolerance for the initial point generator. If this tolerance is not met, `nlpsolve` declares the problem to be infeasible and returns an empty list. (By default, $\epsilon = 10^{-6}$.)
 - * `nlp_integertolerance= ϵ` for some $\epsilon > 0$. This sets the tolerance for evaluation of integrality constraints. (By default, $\epsilon = 10^{-6}$.)
 - * `assume=asmp`, where *asmp* may be one of the following:
 - `nlp_nonnegative`, which restricts all variables to non-negative values (existing positive lower bounds are kept, however),
 - `nlp_nonnegint` or `nonnegint`, which forces all variables to take nonnegative integer values,
 - `nlp_integer` or `integer`, which forces all variables to take integer values,
 - `nlp_binary`, which forces all variables to take 0-1 values.
- `nlpsolve(obj⟨constr,bd,opt⟩)` returns a list [*optobj*,*optdec*], where *optobj* is the optimal value of *obj* and *optobj* is a list of optimal values of the decision variables. If the optimization fails, then *optobj* is the warning message and *optdec* is the last point on the path the solver takes in attempt to find the optimal objective value. If the problem is infeasible or if the solver fails to find a feasible point, then an empty list is returned.
 - `nlpsolve` is a local-search based solver which acts as a global optimizer when the initial point is not set. However, since it is based on a random-sampling heuristic, the global optimality of the solution is not guaranteed. For solving mixed-integer nonlinear problems (MINLP), the branch&bound method is used. Note that `nlpsolve` generally works best for problems with a small number of variables (say, less than ten).

Examples.

- The continuous function f defined by

$$f(x) = \min\{\sqrt{|x+4|} - 1, \sqrt{|x+1|} - 1.005, \sqrt{|x-3|} + 0.5\},$$

has a unique global minimum in $[-5, 5]$ at $x = -1$, with $f(-1) = 1.005$.

Input:

```
nlpsolve(min(sqrt(abs(x+4))-1,sqrt(abs(x+1))-1.005,
            sqrt(abs(x-3))+0.5),x=-5..5)
```

Output:

$[-1.0049992998, [x = -1.0]]$

- Minimize $z = x_1x_4(x_1 + x_2 + x_3) + x_3$ subject to

$$\begin{aligned} x_1x_2x_3x_4 &\geq 25, \\ x_1^2 + x_2^2 + x_3^2 &= 40, \end{aligned}$$

where $1 \leq x_i \leq 5$, $i = 1, 2, 3, 4$.

Input:

```
nlpsolve(x1*x4*(x1+x2+x3)+x3,
          [x1*x2*x3*x4-25>=0,x1^2+x2^2+x3^2+x4^2-40=0],
          [x1,x2,x3,x4]=1..5)
```

Output:

$[17.0140172892, [x_1 = 1.0, x_2 = 4.74299973362,$
 $x_3 = 3.82114985829, x_4 = 1.3794083106]]$

- Minimize $z = (x_1 - 1)^2 + (x_1 - x_2)^2 + (x_3 - 1)^2 + (x_4 - 1)^4 + (x_5 - 1)^6$ subject to

$$\begin{aligned} x_1^2x_4 + \sin(x_4 - x_5) &= 2\sqrt{2}, \\ x_2 + x_3^4x_4^2 &= 8 + \sqrt{2}. \end{aligned}$$

Input:

```
nlpsolve((x1-1)^2+(x1-x2)^2+(x3-1)^2+(x4-1)^4+(x5-1)^6,
          [x1^2*x4+sin(x4-x5)=2*sqrt(2),x2+x3^4*x4^2=8+sqrt(2)])
```

Output:

$[0.241505128809, [x_1 = 1.16617119669, x_2 = 1.18211040318, x_3 = 1.3802572722,$
 $x_4 = 1.50603586392, x_5 = 0.610913318325]]$

- Maximize $z = 3x_1x_2 - x_1 + 6x_2$ subject to $5x_1x_2 - 4x_1 - 4.5x_2 \leq 32$, where $1 \leq x_1, x_2 \leq 5$ are integers.

Input:

```
nlpsolve(3x1*x2-x1+6x2,[5x1*x2-4x1-4.5x2<=32],
[x1,x2]=1..5,assume=integer,maximize)
```

Output:

$[58.0, [x_1 = 2.0, x_2 = 5.0]]$

- Maximize $z = 2x_1 + 3x_2 + 6x_3 - 2x_1x_2 - x_1x_3 - 4x_2x_3$ where all variables are binary.

Input:

```
nlpsolve(2x1+3x2+6x3-2x1*x2-x1*x3-4x2*x3,
assume=nlp_binary,maximize)
```

Output:

$[7.0, [x_1 = 1.0, x_2 = 0, x_3 = 1.0]]$

- Minimize $z = 5y - 2 \ln(x + 1)$ subject to

$$\begin{aligned} e^{x/2} - \frac{1}{2}\sqrt{y} &\leq 1, \\ -2 \ln(x + 1) - y + 2.5 &\leq 0, \\ x + y &\leq 4, \end{aligned}$$

where $x \in [0, 2]$ and $y \in [1, 3]$ is integer.

Input:

```
nlpsolve(5y-2*ln(x+1),
[exp(x/2)-sqrt(y)/2-1<=0,-2*ln(x+1)-y+2.5<=0,x+y-4<=0],
x=0..2,y=1..3,nlp_integervariables=[y])
```

Output:

$[8.54528930252, [x = 1.06959999348, y = 2.0]]$

- Minimize $z = x_1^2 + x_2^2 + 3x_3^2 + 4x_4^2 + 2x_5^2 - 8x_1 - 2x_2 - 3x_3 - x_4 - 2x_5$ subject to

$$\begin{aligned} x_1 + x_2 + x_3 + x_4 + x_5 &\leq 400, \\ x_1 + 2x_2 + 2x_3 + x_4 + 6x_5 &\leq 800, \\ 2x_1 + x_2 + 6x_3 &\leq 200, \\ x_3 + x_4 + 5x_5 &\leq 200, \\ x_1 + x_2 + x_3 + x_4 + x_5 &\geq 55, \\ x_1 + x_2 + x_3 + x_4 &\geq 48, \\ x_2 + x_4 + x_5 &\geq 34, \\ 6x_1 + 7x_5 &\geq 104, \end{aligned}$$

where $0 \leq x_i \leq 99$ are integers for $i = 1, \dots, 5$.

Input:

```

obj:=x1^2+x2^2+3x3^2+4x4^2+2x5^2-8x1-2x2-3x3-x4-2x5;
constr:=[x1+x2+x3+x4+x5<=400,x1+2x2+2x3+x4+6x5<=800,
        2x1+x2+6x3<=200,x3+x4+5x5<=200,
        x1+x2+x3+x4+x5>=55,x1+x2+x3+x4>=48,
        x2+x4+x5>=34,6x1+7x5>=104];
nlpolve(obj,constr,[x1,x2,x3,x4,x5]=0..99,assume=integer)

```

Output:

[807.0, [x₁ = 16.0, x₂ = 22.0, x₃ = 5.0, x₄ = 5.0, x₅ = 7.0]]

6.53.4 Minimax polynomial approximation: minimax

The `minimax` command finds the minimax polynomial approximation of a continuous function on a bounded interval using the Remez algorithm.

- `minimax` takes three mandatory argument and one optional argument:
 - *expr*, a univariate expression representing a continuous function.
 - *var=a..b*, a variable and the interval.
 - *n*, a positive integer, the degree of the resulting polynomial.
 - Optionally *limit=m*, where *m* is a positive integer specifying the number of iterations of the Remez algorithm. (By default, the number of iterations is unlimited.)
- `minimax(expr,var=a..b,n<limit=m)` returns the minimax polynomial approximation of degree *n* or lower that approximates *expr* on *[a,b]*. The largest absolute error of the resulting polynomial will be printed in the message area.

Since the coefficients of *p* are computed numerically, you should avoid setting *n* unnecessary high as it may result in a poor approximation due to the roundoff errors.

Example.

Input:

```
minimax(sin(x),x=0..2*pi,10)
```

Output:

```

max. absolute error: 5.85231264871e-06
5.85141928628 × 10-6 + 0.999777263417x + 0.001400152516x2
- 0.170089663468x3 + 0.0042684302281x4 + 0.00525794778108x5
+ 0.00135760211866x6 - 0.000570502070425x7
+ (6.0729711779 × 10-5) x8 - (2.14787415748 × 10-6) x9
- (1.49765719172 × 10-15) x10

```


6.54 Quadratic forms

6.54.1 Matrix of a quadratic form: q2a

The `q2a` command finds the matrix corresponding to a quadratic form.

- `q2a` takes two arguments:
 - q , the symbolic expression of a quadratic form q .
 - $vars$, a vector of variable names.
- `q2a($q, vars$)` returns the matrix of the quadratic form q .

Example.

Input:

```
q2a(2*x*y, [x, y])
```

Output:

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

6.54.2 Transforming a matrix into a quadratic form: a2q

The `a2q` command finds the quadratic form corresponding to a symmetric matrix.

- `a2q` takes two arguments:
 - A , a symmetric matrix of a quadratic form.
 - $vars$, a vector of variable names whose size is the same as the number of rows of A .
- `a2q($A, vars$)` returns the symbolic expression for the quadratic form.

Examples.

- *Input:*

```
a2q([[0, 1], [1, 0]], [x, y])
```

Output:

$$2xy$$

- *Input:*

```
a2q([[1, 2], [2, 4]], [x, y])
```

Output:

$$x^2 + 4xy + 4y^2$$

6.54.3 Reducing a quadratic form: `gauss`

The `gauss` command uses Gauss's algorithm to write a quadratic form as a sum or difference of squares.

- `gauss` takes two arguments:
 - q , a symbolic expression representing a quadratic form.
 - $vars$, a vector of variable names.
- `gauss($q, vars$)` returns q written as sum or difference of squares.

Example.

Input:

```
gauss(2*x*y, [x,y])
```

Output:

$$\frac{(x+y)^2}{2} - \frac{(-x+y)^2}{2}$$

6.54.4 The conjugate gradient algorithm: `conjugate_gradient`

The `conjugate_gradient` command uses the conjugate gradient algorithm to solve a linear system of equations.

- `conjugate_gradient` takes two mandatory arguments and two optional arguments.
 - A , an $n \times n$ positive definite symmetric matrix.
 - y , a vector of length n .
 - Optionally, x_0 , a vector of length n , an initial approximation.
 - ϵ , a positive number (by default `epsilon`, see Section 3.5.7 p.73, item 9).
- `conjugate_gradient($A, y \langle x_0, \epsilon \rangle$)` returns the solution to $Ax = y$ to within ϵ .

Examples.

- *Input:*

```
conjugate_gradient([ [2,1], [1,5] ], [1,0])
```

Output:

$$\left[\frac{5}{9}, -\frac{1}{9} \right]$$

- *Input:*

```
conjugate_gradient([ [2,1], [1,5] ], [1,0], [0.55, -0.11], 1e-2)
```

Output:

$[0.555, -0.11]$

- *Input:*

```
conjugate_gradient([[2,1],[1,5]],[1,0],[0.55,-0.11],1e-10)
```

Output:

$[0.55555555555556, -0.11111111111111]$

6.54.5 Gram-Schmidt orthonormalization: `gramschmidt`

The `gramschmidt` command uses the Gram-Schmidt procedure to find an orthonormal set of vectors with the same span as a given set.

- `gramschmidt` takes one or two arguments.

With one argument:

- A , a matrix viewed as a list of row vectors or a list of elements that is a basis of a vector subspace.
- f , a function that defines a scalar product on this vector space. If A is a matrix, then this is optional, and by default will be the standard scalar product.

- `gramschmidt(A)` or `gramschmidt(L, f)` returns an orthonormal basis for this scalar product.

Examples.

- *Input:*

```
normal(gramschmidt([[1,1,1],[0,0,1],[0,1,0]]))
```

or:

```
normal(gramschmidt([[1,1,1],[0,0,1],[0,1,0]],dot))
```

Output:

$$\begin{bmatrix} \frac{\sqrt{3}}{3} & \frac{\sqrt{3}}{3} & \frac{\sqrt{3}}{3} \\ -\frac{\sqrt{6}}{6} & -\frac{\sqrt{6}}{6} & \frac{\sqrt{6}}{3} \\ -\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} & 0 \end{bmatrix}$$

- Define a scalar product on the vector space of polynomials by:

$$P \cdot Q = \int_{-1}^1 P(x)Q(x)dx$$

Input:

```
p_scal(p,q):=integrate(p*q,x,-1,1)
gramschmidt([1,1+x],p_scal)
```

or:

```
gramschmidt([1,1+x],(p,q)->integrate(p*q,x,-1,1))
```

Output:

$$\left[\frac{1}{\sqrt{2}}, \frac{1+x-1}{\frac{\sqrt{6}}{3}} \right]$$

6.54.6 Graph of a conic: conic

The `conic` command draws a conic.

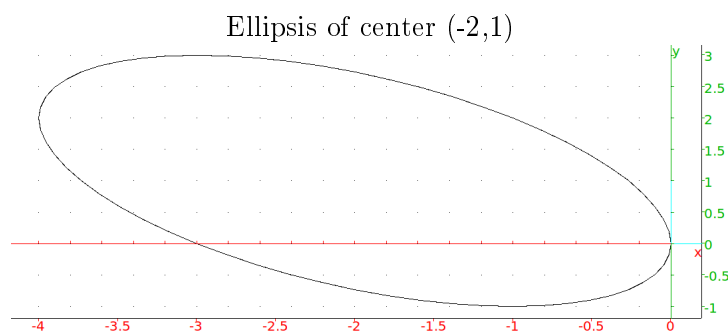
- `conic` takes one mandatory argument and one or two optional arguments:
 - `eq`, the equation of a conic.
 - Optionally, `vars`, a list of the variables (by default $[x, y]$). The variables can also be given as two separate arguments.
- `conic(eq, vars)` draws the conic.

Example.

Input:

```
conic(2*x^2+2*x*y+2*y^2+6*x)
```

Output:



See also the next section for the parametric equation of the conic.

6.54.7 Conic reduction: reduced_conic

The `reduced_conic` command finds the reduced equation of a conic.

- `reduced_conic` takes two arguments:
 - `eq`, the equation of a conic.
 - `vars`, a list of the variable names.
- `reduced_conic(eq, vars)` returns a list whose elements are:
 - the origin of the conic,

- the matrix of a basis in which the conic is reduced,
- 0 or 1 (0 if the conic is degenerate),
- the reduced equation of the conic
- a vector of its parametric equations.

Example.*Input:*

```
reduced_conic(2*x^2+2*x*y+2*y^2+5*x+3,[x,y])
```

Output:

$$\left[\begin{bmatrix} -\frac{5}{3}, \frac{5}{6} \end{bmatrix}, \begin{bmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{bmatrix}, 1, 3x^2 + y^2 - \frac{7}{6}, \right. \\ \left. \left[\frac{-10+5i}{6} + \left(\frac{\sqrt{2}}{2} + \frac{1}{2}i\sqrt{2} \right) \left(\frac{3}{18}\sqrt{14}\cos t + \frac{1}{6}i\sqrt{42}\sin t \right), \right. \right. \\ \left. \left. t, 0, 2\pi, \frac{2}{60}\pi, 2x^2 + 2xy + 2y^2 + 5x + 3, \right. \right. \\ \left. \left. \frac{-10+5i}{6} + \frac{\left(\frac{\sqrt{2}}{2} + \frac{1}{2}i\sqrt{2} \right) \left(\frac{3}{18}\sqrt{14}(1-t^2) + \frac{2}{6}i\sqrt{42}t \right)}{1+t^2} \right] \right]$$

Which means that the conic is not degenerate, its reduced equation is

$$3x^2 + y^2 - 7/6 = 0$$

its origin is $-5/3 + 5i/6$, its axes are parallel to the vectors $(-1, 1)$ and $(-1, -1)$, and its parametric equation is

$$\frac{-10+5i}{6} + \frac{(1+i)}{\sqrt{2}} * \frac{(\sqrt{14} * \cos(t) + i * \sqrt{42} * \sin(t))}{6}$$

where the suggested parameter values for drawing are t from 0 to 2π with `tstep= 2π/60`.

Remark:

Note that if the conic is degenerate and is made of 1 or 2 line(s), the lines are not given by their parametric equation but by the list of two points of the line.

Example.*Input:*

```
reduced_conic(x^2-y^2+3*x+y+2)
```

Output:

$$\left[\begin{bmatrix} -\frac{3}{2}, \frac{1}{2} \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, 0, x^2 - y^2, \begin{bmatrix} \frac{-3+i}{2} & \frac{-1+3i}{2} \\ \frac{-3+i}{2} & \frac{-1-i}{2} \end{bmatrix} \right]$$

6.54.8 Graph of a quadric: `quadric`

The `quadric` command draws a quadric.

- `quadric` takes one mandatory argument and one optional argument:
 - q , the expression of a quadric.
 - Optionally, $vars$, a list of three variable names (by default, $[x, y, z]$). These names can also be given a separate arguments.
- `quadric(q $\langle vars \rangle$)` draws this quadric.

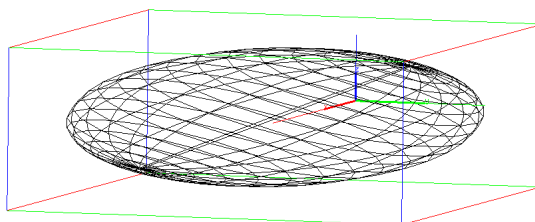
Example.

Input:

```
quadric(7*x^2+4*y^2+4*z^2+4*x*y-4*x*z-2*y*z-4*x+5*y+4*z-18)
```

Output:

Ellipsoid of center $[0.407407407407, -0.962962962963, -0.537037037037]$



See also the next section for the parametric equation of the quadric.

6.54.9 Quadric reduction: `reduced_quadric`

The `reduced_quadric` command finds the reduced equation of a quadric.

- `reduced_quadric` takes two arguments:
 - eq , the equation of a quadric.
 - $vars$, a vector of variable names.
- `reduced_quadric($eq, vars$)` returns a list whose elements are:
 - the origin,
 - the matrix of a basis where the quadric is reduced,
 - 0 or 1 (0 if the quadric is degenerate),
 - the reduced equation of the quadric
 - a vector with its parametric equations.

Warning ! u, v will be used as parameters of the parametric equations: these variables should not be assigned (**purge** them before calling `reduced_quadric`).

Example.

Input:

reduced_quadric(7*x^2+4*y^2+4*z^2+
4*x*y-4*x*z-2*y*z-4*x+5*y+4*z-18)

Output:

$$\left[\left[\frac{11}{27}, -\frac{26}{27}, -\frac{29}{54} \right], \left[\begin{array}{ccc} \frac{\sqrt{6}}{3} & \frac{\sqrt{5}}{5} & -\frac{\sqrt{30}}{15} \\ \frac{\sqrt{6}}{6} & 0 & \frac{\sqrt{30}}{6} \\ -\frac{\sqrt{6}}{6} & \frac{2}{5}\sqrt{5} & \frac{\sqrt{30}}{30} \end{array} \right], [9, 3, 3], 1, 9x^2 + 3y^2 + 3z^2 - \frac{602}{27}, \right. \\ \left[\frac{9\sqrt{6}\sqrt{1806}\sin u \cdot \cos v}{3 \cdot 243} + \frac{9\sqrt{5}\sqrt{602}\sin u \cdot \sin v}{5 \cdot 81} - \frac{9\sqrt{30}\sqrt{602}\cos u}{15 \cdot 81} + \frac{11}{27}, \right. \\ \frac{9\sqrt{6}\sqrt{1806}\sin u \cdot \cos v}{6 \cdot 243} + \frac{9\sqrt{30}\sqrt{602}\cos u}{6 \cdot 81} - \frac{26}{27}, \\ \left. - \frac{9\sqrt{6}\sqrt{1806}\sin u \cdot \cos v}{6 \cdot 243} + \frac{9 \cdot 2\sqrt{5}\sqrt{602}\sin u \cdot \sin v}{5 \cdot 81} + \frac{9\sqrt{30}\sqrt{602}\cos u}{30 \cdot 81} - \frac{29}{54} \right], \\ \left. u = 0 \dots \pi, v = 0 \dots 2\pi, \text{ustep} = \frac{\pi}{20}, \text{vstep} = \frac{2}{20}\pi \right]$$

The output is a list containing:

- The origin (center of symmetry) of the quadric

$$\left[\frac{11}{27}, -\frac{26}{27}, -\frac{29}{54} \right]$$

- The matrix of the basis change:

$$\left[\begin{array}{ccc} \frac{\sqrt{6}}{3} & \frac{\sqrt{5}}{5} & -\frac{\sqrt{30}}{15} \\ \frac{\sqrt{6}}{6} & 0 & \frac{\sqrt{30}}{6} \\ -\frac{\sqrt{6}}{6} & \frac{2}{5}\sqrt{5} & \frac{\sqrt{30}}{30} \end{array} \right],$$

- 1, hence the quadric is not degenerated
- the reduced equation of the quadric:

$$9x^2 + 3y^2 + 3z^2 - \frac{602}{27}$$

- The parametric equations (in the original frame):

$$\left[\left[\frac{9\sqrt{6}\sqrt{1806}\sin u \cdot \cos v}{3 \cdot 243} + \frac{9\sqrt{5}\sqrt{602}\sin u \cdot \sin v}{5 \cdot 81} - \frac{9\sqrt{30}\sqrt{602}\cos u}{15 \cdot 81} + \frac{11}{27}, \right. \right. \\ \frac{9\sqrt{6}\sqrt{1806}\sin u \cdot \cos v}{6 \cdot 243} + \frac{9\sqrt{30}\sqrt{602}\cos u}{6 \cdot 81} - \frac{26}{27}, \\ \left. - \frac{9\sqrt{6}\sqrt{1806}\sin u \cdot \cos v}{6 \cdot 243} + \frac{9 \cdot 2\sqrt{5}\sqrt{602}\sin u \cdot \sin v}{5 \cdot 81} + \frac{9\sqrt{30}\sqrt{602}\cos u}{30 \cdot 81} - \frac{29}{54} \right], \\ \left. u = 0 \dots \pi, v = 0 \dots 2\pi, \text{ustep} = \frac{\pi}{20}, \text{vstep} = \frac{2}{20}\pi \right]$$

Hence the quadric is an ellipsoid and its reduced equation is:

$$9x^2 + 3y^2 + 3z^2 + (-602)/27 = 0$$

after the change of origin $[11/27, (-26)/27, (-29)/54]$, the matrix of basis change is:

$$\begin{bmatrix} \frac{\sqrt{6}}{3} & \frac{\sqrt{5}}{5} & -\frac{\sqrt{30}}{15} \\ \frac{\sqrt{6}}{6} & 0 & \frac{\sqrt{30}}{6} \\ -\frac{\sqrt{6}}{6} & \frac{2\sqrt{5}}{5} & \frac{\sqrt{30}}{30} \end{bmatrix}$$

Its parametric equation is:

$$\begin{cases} x = \frac{\sqrt{6}\sqrt{\frac{602}{243}}\sin(u)\cos(v)}{3} + \frac{\sqrt{5}\sqrt{\frac{602}{81}}\sin(u)\sin(v)}{5} - \frac{\sqrt{30}\sqrt{\frac{602}{81}}\cos(u)}{15} + \frac{11}{27} \\ y = \frac{\sqrt{6}\sqrt{\frac{602}{243}}\sin(u)\cos(v)}{6} + \frac{\sqrt{30}\sqrt{\frac{602}{81}}\cos(u)}{6} - \frac{26}{27} \\ z = \frac{-\sqrt{6}\sqrt{\frac{602}{243}}\sin(u)\cos(v)}{6} + \frac{2\sqrt{5}\sqrt{\frac{602}{81}}\sin(u)\sin(v)}{5} + \frac{\sqrt{30}\sqrt{\frac{602}{81}}\cos(u)}{30} - \frac{29}{54} \end{cases}$$

Remark:

Note that if the quadric is degenerate and made of 1 or 2 plane(s), each plane is not given by its parametric equation but by the list of a point of the plane and of a normal vector to the plane.

Example.

Input:

```
reduced_quadric(x^2-y^2+3*x+y+2)
```

Output:

$$\left[\left[-\frac{3}{2}, \frac{1}{2}, 0 \right], \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix}, \right. \\ \left. [0, 1, -1], x^2 - y^2, \right. \\ \left. \left[\text{hyperplan} \left([1, 1, 0], \left[-\frac{3}{2}, \frac{1}{2}, 0 \right] \right), \text{hyperplan} \left([1, -1, 0], \left[-\frac{3}{2}, \frac{1}{2}, 0 \right] \right) \right] \right]$$

6.55 Equations

6.55.1 Defining an equation: equal

The `equal` command creates equations; it is the infix version of `=`.

- `equal` takes two arguments:
 lhs and rhs , the two sides of the equation.
- `equal(lhs, rhs)` returns the equation $lhs=rhs$.

Example.*Input:*

```
equal(2x-1,3)
```

Output:

$$2x - 1 = 3$$

You can also directly write $(2*x-1)=3$.

6.55.2 Transforming an equation into a difference: equal2diff

The `equal2diff` command turns an equation into the difference of the two sides, resulting in an expression assumed to be equal to 0.

- `equal2diff` takes one argument:
 $lhs=rhs$, an equation.
- `equal2diff(lhs=rhs)` returns the difference $lhs - rhs$.

Example.*Input:*

```
equal2diff(2x-1=3)
```

Output:

$$2x - 1 - 3$$

6.55.3 Transforming an equation into a list: equal2list

The `equal2list` command separates the two sides of an equation.

- `equal2list` takes one argument:
 $lhs=rhs$, an equation.
- `equal2list(lhs=rhs)` returns the sequence lhs, rhs .

Example.*Input:*

```
equal2list(2x-1=3)
```

Output:

$$2x - 1, 3$$

6.55.4 The left member of an equation: `left` gauche `lhs`

(See Section 6.3.4 p.123, Section 6.15.3 p.236, Section 6.37.1 p.445, Section 6.38.2 p.448, Section 6.40.6 p.467 and Section 6.55.5 p.618 for other uses of `left` and `right`.)

The `left` command finds the left hand side of an equation.
For this, `lhs` is a synonym for `left`.

- `left` takes one argument:
 $lhs=rhs$, an equation.
- `left(lhs=rhs)` returns lhs .

Example.

Input:

```
left(2x-1=3)
```

or:

```
lhs(2x-1=3)
```

Output:

$$2x - 1$$

6.55.5 The right member of an equation: `right` droit `rhs`

(See Section 6.3.4 p.123, Section 6.15.3 p.236, Section 6.37.1 p.445, Section 6.38.2 p.448, Section 6.40.6 p.467 and Section 6.55.4 p.618 for other uses of `left` and `right`.)

The `right` command finds the right hand side of an equation.
For this, `rhs` is a synonym for `right`.

- `right` takes one argument:
 $lhs=rhs$, an equation.
- `right(lhs=rhs)` returns rhs .

Example.

Input:

```
right(2x-1=3)
```

or:

```
rhs(2x-1=3)
```

Output:

6.55.6 Solving equation(s): `solve` `cSolve`

The `solve` command solves an equation or a system of polynomial equations. In real mode, `solve` returns only real solutions; to have `solve` return the complex solutions, switch to complex mode (e.g. by checking **Complex** in the cas configuration, see Section 3.5.5 p.72).

The `cSolve` command is identical to `solve`, except it returns the complex solutions whether in real mode or complex mode.

To solve an equation:

- `solve` takes one mandatory argument and one optional argument:
 - `eqn`, an equation or expression assumed to be zero.
 - Optionally, `x`, a variable (by default, `x=x`).
- `solve(eqn, x)` returns the solution to the equation.

For trigonometric equations, `solve` returns by default the principal solutions. To have all the solutions check `All_trig_sol` in the cas configuration (see Section 3.5.7 p.73, item 19).

Examples:

- Solve $x^4 - 1 = 3$

Input:

```
solve(x^4-1=3)
```

Output (in real mode):

$$\left[\sqrt{2}, -\sqrt{2} \right]$$

Output (in complex mode):

$$\left[\sqrt{2}, -\sqrt{2}, i\sqrt{2}, -i\sqrt{2} \right]$$

Also:

Input:

```
cSolve(x^4-1=3)
```

Output (in any mode):

$$\left[-\sqrt{2}, \sqrt{2}, -\sqrt{2}i, \sqrt{2}i \right]$$

- Solve $\exp(x) = 2$

Input:

```
solve(exp(x)=2)
```

Output:

$$\left[\ln(2) \right]$$

- Solve $\cos(2 * x) = 1/2$

Input:

`solve(cos(2*x)=1/2)`

Output:

$$\left[-\frac{\pi}{6}, \frac{\pi}{6}\right]$$

Output (with All_trig_sol checked):

$$\left[\frac{6\pi n_0 + \pi}{6}, \frac{6\pi n_0 - \pi}{6}\right]$$

To solve a system of polynomial equations:

- `solve` takes one mandatory argument and one optional argument:
 - *eqns*, a list of polynomial equations.
 - *vars*, a list of variables.
- `solve(eqns,vars)` returns the solutions to the system of equations.

Examples.

- Find x, y such that $x + y = 1, x - y = 0$

Input:

`solve([x+y=1,x-y],[x,y])`

Output:

$$\left[\left[\frac{1}{2}, \frac{1}{2}\right]\right]$$

- Find x, y such that $x^2 + y = 2, x + y^2 = 2$

Input:

`solve([x^2+y=2,x+y^2=2],[x,y])`

Output:

$$\left[[1, 1], [-2, -2], \left[\frac{\sqrt{5}+1}{2}, -\left(\frac{\sqrt{5}+1}{2}\right)^2 + 2\right], \left[\frac{-\sqrt{5}+1}{2}, -\left(\frac{-\sqrt{5}+1}{2}\right)^2 + 2\right]\right]$$

$$[(\text{sqrt}(5)+1)/2, (1-\text{sqrt}(5))/2]]$$

- Find x, y, z such that $x^2 - y^2 = 0, x^2 - z^2 = 0$

Input:

`solve([x^2-y^2=0,x^2-z^2=0],[x,y,z])`

Output:

$$[[x, x, x], [x, -x, -x], [x, x, -x], [x, -x, x]]$$

- Find the intersection of a straight line (given by a list of equations) and a plane.

For example, let D be the straight line with cartesian equations $[y - z = 0, z - x = 0]$ and let P the plane with equation $x - 1 + y + z = 0$. Find the intersection of D and P .

Input:

$$\text{solve}([y-z=0, z-x=0], x-1+y+z=0, [x, y, z])$$

Output:

$$\left[\left[\frac{1}{3}, \frac{1}{3}, \frac{1}{3} \right] \right]$$

- *Input:*

$$\text{cSolve}([-x^2+y=2, x^2+y], [x, y])$$

Output:

$$[[-i, 1], [i, 1]]$$

6.56 Linear systems

The *augmented matrix* of the system $A \cdot X = b$ is either the matrix obtained by gluing the column vector b to the right of the matrix A (as with `border(A, tran(b))`), representing $A \cdot X = b$, or the matrix obtained by gluing the column vector $-b$ to the right of the matrix A , representing $A \cdot x - b = 0$.

6.56.1 Matrix of a system: `syst2mat`

The `syst2mat` command turns a system of linear equations into its augmented matrix. (For this command, the augmented matrix of $Ax = b$ has the column vector $-b$ glued to the right of A .)

- `syst2mat` takes two as arguments:
 - *eqns*, a list of the equations or expressions (assumed to be equal to zero) of a linear system.
 - *vars*, a list of the variable names.
- `syst2mat(eqns, vars)` returns the augmented matrix of the system.

Warning !!!

The variables must be purged before `syst2mat` is called.

Examples.

- *Input:*

```
syst2mat([x+y,x-y-2],[x,y])
```

Output:

$$\begin{pmatrix} 1 & 1 & 0 \\ 1 & -1 & -2 \end{pmatrix}$$

- *Input:*

```
syst2mat([x+y=0,x-y=2],[x,y])
```

Output:

$$\begin{pmatrix} 1 & 1 & 0 \\ 1 & -1 & -2 \end{pmatrix}$$

6.56.2 Gauss reduction of a matrix: `ref`

A matrix A is in row-echelon form if the first non-zero element of each row is 1 and each of these leading 1s is further right than the leading 1s of the preceding rows. Gaussian elimination will transform a matrix into row echelon form, and the row echelon form of the augmented matrix of a system of linear equations has the same set of solutions as the original, but in a form that is simple to solve.

The `ref` command transforms a matrix into a row echelon form of the matrix.

- `ref` takes one argument:
 A , a matrix.
- `ref(A)` returns a row echelon form of a matrix.

`ref` is typically used to solve a linear system of equations written in matrix form.

Example.

Solve the system:

$$\begin{cases} 3x + y &= -2 \\ 3x + 2y &= 2 \end{cases}$$

Input:

```
ref([[3,1,-2],[3,2,2]])
```

Output:

$$\begin{bmatrix} 1 & \frac{1}{3} & -\frac{2}{3} \\ 0 & 1 & 4 \end{bmatrix}$$

Hence the solution is $y = 4$ (from the last row) and $x = -2$ (substitute y in the first row).

6.56.3 Gauss-Jordan reduction: `rref gaussjord`

The reduced row echelon form of a matrix is the row echelon form (see previous section) with 0s above the leading 1s in each row. Gauss-Jordan reduction will turn any matrix into reduced row echelon form, and the reduced row echelon form of a matrix is unique. If the matrix is the augmented matrix of a system, then the reduced row echelon form of the matrix is the simplest form to solve the system. The `rref` command finds the reduced row echelon form of a matrix (see also Section 6.34.17 p.428).

- `rref` takes one mandatory and one optional argument:
 - A , a matrix.
 - Optionally, n , a positive integer.
- `rref(A(n))` returns the reduced row echelon form of the matrix. With a second argument of n , Gauss-Jordan reduction will be performed on (at most) the first n columns.

Examples.

- Solve the system:

$$\begin{cases} 3x + y &= -2 \\ 3x + 2y &= 2 \end{cases}$$

Input:

```
rref([[3,1,-2],[3,2,2]])
```

Output:

$$\begin{bmatrix} 1 & 0 & -2 \\ 0 & 1 & 4 \end{bmatrix}$$

Hence $x = -2$ and $y = 4$ is the solution of this system.

- `rref` can also solve several linear systems of equations having the same matrix of coefficients by augmenting the matrix of coefficients by vectors representing the right hand side of the equations for each equation. For example, Solve the systems:

$$\begin{cases} 3x + y &= -2 \\ 3x + 2y &= 2 \end{cases}$$

and

$$\begin{cases} 3x + y &= 1 \\ 3x + 2y &= 2 \end{cases}$$

Input:

```
rref([[3,1,-2,1],[3,2,2,2]])
```

Output:

$$\begin{bmatrix} 1 & 0 & -2 & 0 \\ 0 & 1 & 4 & 1 \end{bmatrix}$$

Which means that $x = -2$ and $y = 4$ is the solution of the first system and $x = 0$ and $y = 1$ is the solution of the second system.

- *Input:*

```
rref([ [3,1,-2,1], [3,2,2,2] ], 1)
```

Output:

$$\begin{bmatrix} 1 & \frac{1}{3} & -\frac{2}{3} & \frac{1}{3} \\ 0 & 3 & 12 & 3 \end{bmatrix}$$

and Gauss-Jordan reduction has been performed only on the first column.

6.56.4 Solving $AX = b$: `simult`

The `simult` command can solve a linear system of equations or several linear systems of equations with the same matrix of coefficients (see also [6.56.3](#)).

- `simult` takes two arguments:
 - A , a matrix (the matrix of coefficients of a system).
 - b , a column vector (representing the right hand side of the system) or a matrix (where each column represents the right hand side of an equation).
- `simult(A,b)` returns a column vector of the solutions (or a matrix where each column is the column vector of a solution).

Examples.

- Solve

$$\begin{cases} 3x + y &= -2 \\ 3x + 2y &= 2 \end{cases}$$

Input:

```
simult([ [3,1], [3,2] ], [[-2], [2]])
```

Output:

$$\begin{bmatrix} -2 \\ 4 \end{bmatrix}$$

```
[[ -2], [4]]
```

Hence $x = -2$ and $y = 4$ is the solution.

- Solve

$$\begin{cases} 3x + y &= -2 \\ 3x + 2y &= 2 \end{cases}$$

and

$$\begin{cases} 3x + y &= 1 \\ 3x + 2y &= 2 \end{cases}$$

Input:

```
simult([[3,1],[3,2]], [[-2,1],[2,2]])
```

Output:

$$\begin{bmatrix} -2 & 0 \\ 4 & 1 \end{bmatrix}$$

So $x = -2$ and $y = 4$ is the solution of the first system of equations and $x = 0$ and $y = 1$ is the solution of the second system.

6.56.5 Step by step Gauss-Jordan reduction of a matrix: pivot

One step of Gauss-Jordan elimination involves taking a non-zero element of a matrix (the pivot) and adding multiples of its row to the other rows to get zeros above and below the pivot. The `pivot` command performs this operation.

- `pivot` takes three arguments:
 - A , a matrix with n rows and p columns.
 - l and c , two integers such that l is the index of a row of A and c is an index of column of A , and the element of A in row l and column c is non-zero.
- `pivot(A, l, c)` returns the result of performing one step of the Gauss-Jordan method using the element of A in row l , column c as pivot.

Examples.

- *Input:*

```
pivot([[1,2],[3,4],[5,6]],1,1)
```

Output:

$$\begin{bmatrix} -2 & 0 \\ 3 & 4 \\ 2 & 0 \end{bmatrix}$$

- *Input:*

```
pivot([[1,2],[3,4],[5,6]],0,1)
```

Output:

$$\begin{bmatrix} 1 & 2 \\ 2 & 0 \\ 4 & 0 \end{bmatrix}$$

6.56.6 Linear system solving: `linsolve`

The `linsolve` command solves systems of linear equations. It can take its arguments as a list of equations or as a matrix of coefficients followed by a vector of the right hand side. It can also take the matrix of coefficients after an LU factorization (see Section 6.49.5 p.567), which can be useful when you have several systems of equations which only differ on their right hand side. If the **Step by step** box is checked in the general configuration (see Section 3.5.9 p.77), `linsolve` will show you the steps in getting a solution.

Given a system of equations:

- `linsolve` takes two arguments:
 - *eqns*, a list of linear equations or expressions (where each expression is assumed to be equal to zero).
 - *vars*, a list of variable names.
- `linsolve(eqns,vars)` returns the solution of the equations as a list.

Examples.

- *Input:*

```
linsolve([2*x+y+z=1,x+y+2*z=1,x+2*y+z=4],[x,y,z])
```

Output:

$$\left[-\frac{1}{2}, \frac{5}{2}, -\frac{1}{2} \right]$$

Which means that

$$x = -\frac{1}{2}, y = \frac{5}{2}, z = -\frac{1}{2}$$

is the solution of the system:

$$\begin{cases} 2x + y + z = 1 \\ x + y + 2z = 1 \\ x + 2y + z = 4 \end{cases}$$

- *Input:*

```
linsolve([x+2*y+3*z=1,3*x+2*y+z=2],[x,y,z])
```

Output:

$$\left[z + \frac{1}{2}, -2z + \frac{1}{4}, z \right]$$

Given the matrix of coefficients:

- `linsolve` takes two arguments:
 - *A*, the matrix of coefficients of a linear system.
 - *b*, a list of the values of the right hand side of the system.

- `linsolve(A,b)` returns the solution of the corresponding equations as a list.

Example.

Input:

```
linsolve ([[2,1,1], [1,1,2], [1,2,1]], [1,1,4])
```

Output:

$$\left[-\frac{1}{2}, \frac{5}{2}, -\frac{1}{2}\right]$$

If the **Step by step** option is checked in the general configuration, a window

will also pop up showing:

$$\begin{bmatrix} 2 & 1 & 1 & -1 \\ 1 & 1 & 2 & -1 \\ 1 & 2 & 1 & -4 \end{bmatrix}$$

Reduce column 1 with pivot 1 at row 2

Exchange row 1 and row 2

$$L_2 < -(1) * L_2 - (2) * L_1 \text{ on } \begin{bmatrix} 1 & 1 & 2 & -1 \\ 2 & 1 & 1 & -1 \\ 1 & 2 & 1 & -4 \end{bmatrix}$$

$$L_3 < -1 * L_3 - (1) * L_1 \text{ on } \begin{bmatrix} 1 & 1 & 2 & -1 \\ 0 & -1 & -3 & 1 \\ 1 & 2 & 1 & -4 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 1 & 2 & -1 \\ 0 & -1 & -3 & 1 \\ 0 & 1 & -1 & -3 \end{bmatrix}$$

Reduce column 2 with pivot 1 at row 3

Exchange row 2 and row 3

$$L_1 < -(1) * L_1 - (1) * L_2 \text{ on } \begin{bmatrix} 1 & 1 & 2 & -1 \\ 0 & 1 & -1 & -3 \\ 0 & -1 & -3 & 1 \end{bmatrix}$$

$$L_3 < -(1) * L_3 - (-1) * L_2 \text{ on } \begin{bmatrix} 1 & 0 & 3 & 2 \\ 0 & 1 & -1 & -3 \\ 0 & -1 & -3 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 3 & 2 \\ 0 & 1 & -1 & -3 \\ 0 & 0 & -4 & -2 \end{bmatrix}$$

Reduce column 3 with pivot -4 at row 3

$$L_1 < -(1) * L_1 - (-3/4) * L_3 \text{ on } \begin{bmatrix} 1 & 0 & 3 & 2 \\ 0 & 1 & -1 & -3 \\ 0 & 0 & -4 & -2 \end{bmatrix}$$

$$L_2 < -(1) * L_2 - (1/4) * L_3 \text{ on } \begin{bmatrix} 1 & 0 & 0 & \frac{1}{2} \\ 0 & 1 & -1 & -3 \\ 0 & 0 & -4 & -2 \end{bmatrix}$$

$$\text{End reduction } \begin{bmatrix} 1 & 0 & 0 & \frac{1}{2} \\ 0 & 1 & 0 & -\frac{5}{2} \\ 0 & 0 & -4 & -2 \end{bmatrix}$$

$$\begin{bmatrix} 2 & 1 & 1 & -1 \\ 1 & 1 & 2 & -1 \\ 1 & 2 & 1 & -4 \end{bmatrix}$$

Reduce column 1 with pivot 1 at row 2

Exchange row 1 and row 2

$$L_2 < -(1) * L_2 - (2) * L_1 \text{ on } \begin{bmatrix} 1 & 1 & 2 & -1 \\ 2 & 1 & 1 & -1 \\ 1 & 2 & 1 & -4 \end{bmatrix}$$

$$L_3 < -(1) * L_3 - (1) * L_1 \text{ on } \begin{bmatrix} 1 & 1 & 2 & -1 \\ 0 & -1 & -3 & 1 \\ 1 & 2 & 1 & -4 \end{bmatrix}$$

Given the matrix of coefficients in factored form:

- `linsolve` takes four arguments:
 - P, L, U , the LU decomposition of the matrix of coefficients.
 - b , a list of the values of the right hand side of the system.
- `linsolve(P, L, U, b)` returns the solution of the corresponding equations as a list.

Example.

Input:

```
p,l,u:=lu([[2,1,1],[1,1,2],[1,2,1]])
linsolve(p,l,u,[1,1,4])
```

Output:

$$\left[-\frac{1}{2}, \frac{5}{2}, -\frac{1}{2}\right]$$

The `linsolve` command also solves systems with coefficients in $\mathbb{Z}/n\mathbb{Z}$.

Example.

Input:

```
linsolve([2*x+y+z-1,x+y+2*z-1,x+2*y+z-4]%3,[x,y,z])
```

Output:

$$[1 \% 3, 1 \% 3, 1 \% 3]$$

6.56.7 Solving a linear system using the Jacobi iteration method: `jacobi_linsolve`

The `jacobi_linsolve` command finds the solution of a linear system of equations using the Jacobi iteration method.

- `jacobi_linsolve` command takes two mandatory arguments and two optional arguments:
 - A , the matrix of coefficients of a system.
 - b , the right hand side of the system as a list.
 - Optionally, m , an integer indicating the maximum number of iterations (by default $m = \text{maxiter}$, see Section 3.5.7 p.73, item 6).
 - Optionally, ϵ , a positive number indicating the error tolerance (by default $\epsilon = \text{epsilon}$, see Section 3.5.7 p.73, item 9).

`jacobi_linsolve($A, b(m, \epsilon)$)` returns the solution of the system.

Examples.

- *Input:*

```
A:=[[100,2],[2,100]];
jacobi_linsolve(A,[0,1],1e-12);
```

Output:

```
[-0.000200080032, 0.0100040016006]
```

- *Input:*

```
evalf(linsolve(A,[0,1]))
```

Output:

```
[-0.000200080032013, 0.0100040016006]
```

6.56.8 Solving a linear system using the Gauss-Seidel iteration method: `gauss_seidel_linsolve`

The `gauss_seidel_linsolve` command finds the solution of a linear system of equations using the Gauss-Seidel method.

- `gauss_seidel_linsolve` command takes two mandatory arguments and three optional arguments (including an optional first argument):
 - Optionally, ω , used for a general form of the Gauss-Seidel method (the successive overrelaxation method) (by default, $\omega = 1$).
 - A , the matrix of coefficients of a system.
 - b , the right hand side of the system as a list.
 - Optionally, ϵ , a positive number indicating the error tolerance (by default $\epsilon = \text{epsilon}$, see Section 3.5.7 p.73, item 9).
 - Optionally, m , an integer indicating the maximum number of iterations (by default $m = \text{maxiter}$, see Section 3.5.7 p.73, item 6).

`gauss_seidel_linsolve(A,b[ϵ ,m])` returns the solution of the system.

Examples.

- *Input:*

```
A:=[[100,2],[2,100]];
gauss_seidel_linsolve(A,[0,1],1e-12);
```

Output:

```
[-0.000200080032013, 0.0100040016006]
```

- *Input:*

```
gauss_seidel_linsolve (1.5, A, [0,1], 1e-12);
```

Output:

```
[-0.000200080032218, 0.0100040016006]
```

6.56.9 The least squares solution of a linear system: LSQ lsq

The `lsq` command finds the least squares solution to a matrix equation $AX = b$.

LSQ is a synonym for `lsq`.

- `lsq` takes two arguments:
 - A , a matrix.
 - b , a vector or matrix
- `lsq(A,b)` returns the least squares solution to the equation $AX = b$.

Examples.

- *Input:*

```
lsq([[1,2],[3,4]], [5,11])
```

Output:

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

- *Input:*

```
lsq([[1,2],[3,4]], [[5,7],[11,9]])
```

Output:

$$\begin{bmatrix} 1 & -5 \\ 2 & 6 \end{bmatrix}$$

- Note that:

Input:

```
linsolve([[1,2],[3,4],[3,6]]*[x, y] - [5,11,13],[x, y])
```

Output:

$$\square$$

since the linear system has no solution. You can still find the least squares solution:

Input:

```
lsq([[1,2],[3,4],[3,6]], [5,11,13])
```

Output:

$$\begin{bmatrix} \frac{11}{5} \\ \frac{11}{10} \end{bmatrix}$$

- The least squares solution:

Input:

```
lsq([[3,4]], [12])
```

Output:

$$\begin{bmatrix} \frac{36}{25} \\ \frac{48}{25} \end{bmatrix}$$

represents the point on the line $3x + 4y = 12$ closest to the origin;

Input:

```
coordinates(projection(line(3*x+4*y=12),point(0)))
```

(see Section 13.13.4 p.935, Section 13.15.8 p.959, Section 8.7.1 p.676 and Section 13.6.2 p.886)

Output:

$$\left[\frac{36}{25}, \frac{48}{25} \right]$$

6.56.10 Finding linear recurrences: reverse_resolve

The **reverse_resolve** command finds a linear recurrence relation given the first few terms.

- **reverse_resolve** takes one argument:
 $v = [v_0, \dots, v_{2n-1}]$, a vector made of the first $2n$ terms of a sequence (v_n) which is supposed to verify a linear recurrence relation of degree smaller than n

$$x_n * v_{n+k} + \dots + x_0 * v_k = 0$$

where the x_j are $n + 1$ unknowns.

- **reverse_resolve**(v) returns the list $x = [x_n, \dots, x_0]$ of the x_j coefficients (if $x_n \neq 0$ it is reduced to 1).

In other words, **reverse_resolve** solves the linear system of n equations:

$$\begin{array}{rcl} x_n * v_n + \dots + x_0 * v_0 & = & 0 \\ & \dots & \\ x_n * v_{n+k} + \dots + x_0 * v_k & = & 0 \\ & \dots & \\ x_n * v_{2n-1} + \dots + x_0 * v_{n-1} & = & 0 \end{array}$$

The matrix A of the system has n rows and $n + 1$ columns:

$$A = \begin{pmatrix} v_n & \dots & v_0 & 0 \\ \vdots & & & \vdots \\ v_{n+k} & \dots & v_k & 0 \\ \vdots & & & \vdots \\ v_{2n-1} & v_2 & v_{n-1} & 0 \end{pmatrix}$$

`reverse_resolve` returns the list $x = [x_n, \dots, x_1, x_0]$ with $x_n = 1$ and x is the solution of the system $Ax = 0$.

Examples:

- Find a sequence satisfying a linear recurrence of degree at most 2 whose first elements are 1, -1, 3, 3.

Input:

```
reverse_resolve([1, -1, 3, 3])
```

Output:

```
[1, -3, -6]
```

Hence $x_0 = -6$, $x_1 = -3$, $x_2 = 1$ and the recurrence relation is

$$v_{k+2} - 3v_{k+1} - 6v_k = 0$$

- Find a sequence satisfying a linear recurrence of degree at most 3 whose first elements are 1, -1, 3, 3, -1, 1.

Input:

```
reverse_resolve([1, -1, 3, 3, -1, 1])
```

Output:

```
[1, -1/2, 1/2, -1]
```

Hence, $x_0 = -1$, $x_1 = 1/2$, $x_2 = -1/2$, $x_3 = 1$, the recurrence relation is

$$v_{k+3} - \frac{1}{2}v_{k+2} + \frac{1}{2}v_{k+1} - v_k = 0$$

6.57 Differential equations

This section is limited to symbolic (or exact) solutions of differential equations. For numeric solutions of differential equations, see `odesolve` (Section 10.3.5 p.812). For graphic representation of solutions of differential equations, see `plotfield` (Section 8.18 p.698), `plotode` (Section 8.19 p.699) and `interactive_plotode` (Section 8.20 p.701).

6.57.1 Solving differential equations: `desolve` `deSolve` `dsolve`

The `desolve` (or `deSolve`) command can solve:

- linear differential equations with constant coefficients,
- first order linear differential equations,
- first order differential equations without y ,
- first order differential equations without x ,

- first order differential equations with separable variables,
- first order homogeneous differential equations ($y' = F(y/x)$),
- first order differential equations with integrating factor,
- first order Bernoulli differential equations ($a(x)y' + b(x)y = c(x)y^n$),
- first order Clairaut differential equations ($y = x * y' + f(y')$).

deSolve is a synonym for **desolve**.

- **desolve** takes one mandatory arguments and two optional arguments:
 - *de*, a differential equation or list of differential equations, including any initial conditions.
 - Optionally, *x*, the variable (by default **x**).
 - Optionally, *y*, the unknown function (by default **y**). The unknown function can be given in variable form (such as **y**) or function form (such as **y(x)**), in which case the variable doesn't have to be given as a separate argument.
- **desolve(de <, x, y)** returns the solution of the differential equation.

In the differential equations, the function *y* can be denoted by *y* or *y(x)*, the derivative by *y'*, *y'(x)* or **diff(y(x), x)**, etc.

Examples.

- *Input:*

```
desolve(y''+2*y'+y,y)
```

Output:

$$e^{-x}(c_0x + c_1)$$

- *Input:*

```
desolve([y''+2*y'+y,y(0)=1,y'(0)=0],y)
```

Output:

$$e^{-x}(x + 1)$$

- *Input:*

```
desolve(diff(y(t),t$2)+2*diff(y(t),t)+y(t),y(t))
```

or:

```
desolve(diff(y(t),t$2)+2*diff(y(t),t)+y(t),t,y)
```

Output:

$$e^{-t}(c_0t + c_1)$$

- *Input:*

```
desolve([diff(y(t),t$2)+2*diff(y(t),t)+y(t),y(0)=1,y'(0)=0],y(t))
```

or:

```
desolve([diff(y(t),t$2)+2*diff(y(t),t)+y(t),y(0)=1,y'(0)=0],t,y)
```

Output:

$$e^{-t}(t+1)$$

- *Solve:*

$$y'' + y = \cos(x)$$

Input (typing twice prime for y''):

```
desolve(y''+y=cos(x),y)
```

or:

```
desolve((diff(diff(y))+y)=(cos(x)),y)
```

Output:

$$c_0 \cos x + c_1 \sin x + \frac{2x \sin x + \cos x}{4}$$

c_0, c_1 are the constants of integration: $y(0) = c_0$ $y'(0) = c_1$.

- If the variable is not x but t :

Input:

```
desolve(derive(derive(y(t),t),t)+y(t)=cos(t),t,y)
```

Output:

$$c_0 \cos t + c_1 \sin t + \frac{2t \sin t + \cos t}{4}$$

c_0, c_1 are the constants of integration: $y(0) = c_0$ and $y'(0) = c_1$.

- *Solve:*

$$y'' + y = \cos(x), \quad y(0) = 1$$

Input:

```
desolve([y''+y=cos(x),y(0)=1],y)
```

Output:

$$\frac{3}{4} \cos x + c_1 \sin x + \frac{2x \sin x + \cos x}{4}$$

- Solve:

$$y'' + y = \cos(x) \quad (y(0))^2 = 1$$

Input:

```
desolve([y''+y=cos(x),y(0)^2=1],y)
```

Output:

$$\left[\frac{3 \cos x}{4} + c_1 \sin x + \frac{2x \sin x + \cos x}{4}, -\frac{5 \cos x}{4} + c_1 \sin x + \frac{2x \sin x + \cos x}{4} \right]$$

each component of this list is a solution, you have two solutions depending on the constant c_1 ($y'(0) = c_1$) and corresponding to $y(0) = 1$ and to $y(0) = -1$.

- Solve:

$$y'' + y = \cos(x), \quad (y(0))^2 = 1 \quad y'(0) = 1$$

Input:

```
desolve([y''+y=cos(x),y(0)^2=1,y'(0)=1],y)
```

Output:

$$\left[\frac{3 \cos x}{4} + \sin x + \frac{2x \sin x + \cos x}{4}, -\frac{5 \cos x}{4} + \sin x + \frac{2x \sin x + \cos x}{4} \right]$$

each component of this list is a solution (you have two solutions).

- Solve:

$$y'' + 2y' + y = 0$$

Input:

```
desolve(y''+2*y'+y=0,y)
```

Output:

$$e^{-x} (c_0 x + c_1)$$

the solution depends on two constants of integration: c_0 and c_1 ($y(0) = c_0$ and $y'(0) = c_1$).

- Solve:

$$y'' - 6y' + 9y = x e^{3x}$$

Input:

```
desolve(y''-6*y'+9*y=x*exp(3*x),y)
```

Output:

$$e^{3x} (c_0 x + c_1) + \frac{1}{6} x^3 e^{3x}$$

The solution depends on 2 constants of integration: c_0, c_1 ($y(0) = c_0$ and $y'(0) = c_1$).

- Examples of first order linear equations.

– Solve:

$$xy' + y - 3x^2 = 0$$

Input:

```
desolve(x*y'+y-3*x^2,y)
```

Output:

$$\frac{c_0 + x^3}{x}$$

– Solve:

$$y' + x * y = 0, y(0) = 1$$

Input:

```
desolve([y'+x*y=0, y(0)=1],y)
```

or:

```
desolve((y'+x*y=0) && (y(0)=1),y)
```

Output:

$$e^{-\frac{x^2}{2}}$$

– Solve:

$$x(x^2 - 1)y' + 2y = 0$$

Input:

```
desolve(x*(x^2-1)*y'+2*y=0,y)
```

Output:

$$\frac{c_0 x^2}{x^2 - 1}$$

– Solve:

$$x(x^2 - 1)y' + 2y = x^2$$

Input:

```
desolve(x*(x^2-1)*y'+2*y=x^2,y)
```

Output:

$$\frac{c_0 x^2 + x^2 \ln x}{x^2 - 1}$$

– If the variable is t instead of x , for example, solve:

$$t(t^2 - 1)y'(t) + 2y(t) = t^2$$

Input:

```
desolve(t*(t^2-1)*diff(y(t),t)+2*y(t)=(t^2),y(t))
```

Output:

$$\frac{c_0 t^2 + t^2 \ln t}{t^2 - 1}$$

– Solve:

$$x(x^2 - 1)y' + 2y = x^2, y(2) = 0$$

Input:

```
desolve([x*(x^2-1)*y'+2*y=x^2,y(2)=0],y)
```

Output:

$$\frac{-\ln(2)x^2 + x^2 \ln x}{x^2 - 1}$$

– Solve:

$$\sqrt{1+x^2}y' - x - y = \sqrt{1+x^2}$$

Input:

```
desolve(y'*sqrt(1+x^2)-x-y-sqrt(1+x^2),y)
```

Output:

$$\frac{-c_0 + \ln(\sqrt{x^2 + 1} - x)}{x - \sqrt{x^2 + 1}}$$

- Examples of first differential equations with separable variables.

– Solve:

$$y' = 2\sqrt{y}$$

Input:

```
desolve(y'=2*sqrt(y),y)
```

Output:

$$\left[\left(-\frac{1}{2}c_0 + x \right)^2 \right]$$

– Solve:

$$xy' \ln(x) - y(3 \ln(x) + 1) = 0$$

Input:

```
desolve(x*y'*ln(x)-(3*ln(x)+1)*y,y)
```

Output:

$$c_0 x^3 \ln x$$

- Examples of Bernoulli differential equations $a(x)y' + b(x)y = c(x)y^n$ where n is a real constant.
The method used is to divide the equation by y^n , so that it becomes a first order linear differential equation in $u = 1/y^{n-1}$.

– Solve:

$$xy' + 2y + xy^2 = 0$$

Input:

```
desolve(x*y'+2*y+x*y^2,y)
```

Output:

$$\left[0, -\frac{1}{c_1 x^2 + x}\right]$$

– Solve:

$$xy' - 2y = xy^3$$

Input:

$$\text{desolve}(x*y'-2*y-x*y^3,y)$$

Output:

$$\left[\left(\left(-\frac{1}{5} \cdot 2x^5 + c_0\right) e^{-4 \ln x}\right)^{-\frac{1}{2}}, -\left(\left(-\frac{1}{5} \cdot 2x^5 + c_0\right) e^{-4 \ln x}\right)^{-\frac{1}{2}}\right]$$

– Solve:

$$x^2 y' - 2y = x e^{(4/x)} y^3$$

Input:

$$\text{desolve}(x*y'-2*y-x*\exp(4/x)*y^3,y)$$

Output:

$$\left[\left(\left(-\int 2x^4 \left(e^{\frac{1}{x}}\right)^4 dx + c_0\right) e^{-4 \ln x}\right)^{-\frac{1}{2}}, -\left(\left(-\int 2x^4 \left(e^{\frac{1}{x}}\right)^4 dx + c_0\right) e^{-4 \ln x}\right)^{-\frac{1}{2}}\right]$$

- Examples of first order homogeneous differential equations ($y' = F(y/x)$), the method of integration is to search for $t = y/x$ instead of y .

– Solve:

$$3x^3 y' = y(3x^2 - y^2)$$

Input:

$$\text{desolve}(3*x^3*\text{diff}(y)=((3*x^2-y^2)*y),y)$$

Output:

$$\left[0, -\frac{x\sqrt{6}\sqrt{\ln\left(\frac{x}{c_0}\right)}}{2\ln\left(\frac{x}{c_0}\right)}, \frac{x\sqrt{6}\sqrt{\ln\left(\frac{x}{c_0}\right)}}{2\ln\left(\frac{x}{c_0}\right)}\right]$$

Hence the solutions are $y = 0$ and the family of curves with parametric equations $x = c_0 \exp(3/(2t^2))$, $y = t * c_0 \exp(3/(2t^2))$ (the parameter is denoted by ‘ t ’ in the answer).

- Examples of first order differential equations with an integrating factor. By multiplying the equation by a function of x, y , it becomes a closed differential form.

– Solve:

$$yy' + x = 0$$

Input:

`desolve(y*y'+x,y)`

Output:

$$\left[\sqrt{-x^2 - 2c_0}, -\sqrt{-x^2 - 2c_0} \right]$$

In this example, $xdx + ydy$ is closed, the integrating factor was 1.

– Solve:

$$2xyy' + x^2 - y^2 + a^2 = 0$$

Input:

`desolve(2*x*y*y'+x^2-y^2+a^2,y)`

Output:

$$\left[\sqrt{a^2 - x^2 - c_1x}, -\sqrt{a^2 - x^2 - c_1x} \right]$$

In this example, the integrating factor was $1/x^2$.

- Example of first order differential equations without x .

Solve:

$$(y + y')^4 + y' + 3y = 0$$

This kind of equation cannot be solved directly by **Xcas**, you can use the following steps on solve it with the help of **Xcas**. The idea is to find a parametric representation of $F(u, v) = 0$ where the equation is $F(y, y') = 0$. Let $u = f(t), v = g(t)$ be such a parametrization of $F = 0$, then $y = f(t)$ and $dy/dx = y' = g(t)$. Hence

$$dy/dt = f'(t) = y' * dx/dt = g(t) * dx/dt$$

The solution is the curve of parametric equations $x(t), y(t) = f(t)$, where $x(t)$ is solution of the differential equation $g(t)dx = f'(t)dt$.

Back to the example, you can let $y + y' = t$, hence:

$$y = -t - 8 * t^4, \quad y' = dy/dx = 3 * t + 8 * t^4 \quad dy/dt = -1 - 32 * t^3$$

therefore

$$(3 * t + 8 * t^4) * dx = (-1 - 32 * t^3)dt$$

Input:

`desolve((3*t+8*t^4)*diff(x(t),t)=(-1-32*t^3),x(t))`

Output:

$$\frac{9c_0 - 11 \ln(8t^3 + 3) - \ln(t^3)}{9}$$

The solution is the curve of parametric equation:

$$x(t) = -11 * 1/9 * \ln(8 * t^3 + 3) + 1/9 * \ln(t^3) + c_0, \quad y(t) = -t - 8 * t^4$$

- Examples of first order Clairaut differential equations ($y = x * y' + f(y')$).

The solutions are the lines D_m of equation $y = mx + f(m)$ where m is a real constant.

– Solve:

$$xy' + y'^3 - y = 0$$

Input:

```
desolve(x*y'+y'^3-y,y)
```

Output:

$$[c_0x + c_0^3]$$

– Solve:

$$y - xy' - \sqrt{a^2 + b^2 * y'^2} = 0$$

Input:

```
desolve((y-x*y'-sqrt(a^2+b^2*y'^2),y)
```

Output:

$$\left[c_0x + \sqrt{a^2 + b^2 c_0^2} \right]$$

6.57.2 Laplace transform and inverse Laplace transform: `laplace` `ilaplace` `invlaplace`

Denoting by \mathcal{L} the Laplace transform, you get the following:

$$\begin{aligned}\mathcal{L}(y)(x) &= \int_0^{+\infty} e^{-xu} y(u) du \\ \mathcal{L}^{-1}(g)(x) &= \frac{1}{2i\pi} \int_C e^{zx} g(z) dz\end{aligned}$$

where C is a closed contour enclosing the poles of g .

The `laplace` command finds the Laplace transform of a function.

- `laplace` takes one mandatory argument and two optional arguments:
 - `expr`, an expression involving a variable.
 - Optionally, x , the variable name (by default `x`).
 - Optionally, s , a variable for the output (by default x).
- `laplace(expr(x),)` returns the Laplace transform of `expr`.

Examples.

- *Input:*

```
laplace(sin(x))
```

Output:

$$\frac{1}{x^2 + 1}$$

- *Input:*

```
laplace(sin(t),t)
```

Output:

$$\frac{1}{t^2 + 1}$$

• *Input:*

`laplace(sin(t),t,s)`

Output:

$$\frac{1}{s^2 + 1}$$

The `ilaplace` command finds the Laplace transform of a function. `invlaplace` is a synonym for `ilaplace`.

- `ilaplace` takes one mandatory argument and two optional arguments:
 - *expr*, an expression involving a variable.
 - Optionally, *x*, the variable name (by default `x`).
 - Optionally, *s*, a variable for the output (by default *x*).
- `ilaplace(expr(x))` returns the inverse Laplace transform of *expr*.

The Laplace transform has the following properties:

$$\begin{aligned}\mathcal{L}(y')(x) &= -y(0) + x\mathcal{L}(y)(x) \\ \mathcal{L}(y'')(x) &= -y'(0) + x\mathcal{L}(y')(x) \\ &= -y'(0) - xy(0) + x^2\mathcal{L}(y)(x)\end{aligned}$$

These properties make the Laplace transform and inverse Laplace transform useful for solving linear differential equations with constant coefficients. For example, suppose you have

$$\begin{aligned}y'' + py' + qy &= f(x) \\ y(0) &= a, \quad y'(0) = b\end{aligned}$$

then

$$\begin{aligned}\mathcal{L}(f)(x) &= \mathcal{L}(y'' + py' + qy)(x) \\ &= -y'(0) - xy(0) + x^2\mathcal{L}(y)(x) - py(0) + px\mathcal{L}(y)(x) + q\mathcal{L}(y)(x) \\ &= (x^2 + px + q)\mathcal{L}(y)(x) - y'(0) - (x + p)y(0)\end{aligned}$$

Therefore, if $a = y(0)$ and $b = y'(0)$, you get

$$\mathcal{L}(f)(x) = (x^2 + px + q)\mathcal{L}(y)(x) - (x + p)a - b$$

and the solution of the differential equation is:

$$y(x) = \mathcal{L}^{-1}((\mathcal{L}(f)(x) + (x + p)a + b)/(x^2 + px + q))$$

Example.

Solve:

$$y'' - 6y' + 9y = xe^{3x}, \quad y(0) = c_0, \quad y'(0) = c_1$$

Here, $p = -6$, $q = 9$.

Input:

```
laplace(x*exp(3*x))
```

Output:

$$\frac{1}{x^2 - 6x + 9}$$

Input:

```
ilaplace((1/(x^2-6*x+9)+(x-6)*c_0+c_1)/(x^2-6*x+9))
```

Output:

$$\frac{1}{6} (x^3 - 18xc_0 + 6xc_1 + 6c_0) e^{3x}$$

Note that this equation could be solved directly.

Input:

```
desolve(y''-6*y'+9*y=x*exp(3*x),y)
```

Output:

$$e^{3x} (c_0x + c_1) + \frac{1}{6} x^3 e^{3x}$$

You also can use the `addtable` command Laplacians of unspecified functions (see Section 6.26.2 p.336).

6.57.3 Solving linear homogeneous second-order ODE with rational coefficients: `kovacicsols`

The `kovacicsols` command finds Liouvillian solutions of ordinary linear homogeneous second-order differential equations of the form

$$a y'' + b y' + c y = 0, \quad (6.12)$$

where a , b and c are rational functions of the independent variable. `kovacicsols` uses Kovacic's algorithm.

- `kovacicsols` takes one mandatory argument and two optional arguments:
 - *kode*, an equality of the form of equation (6.12), an expression for the left-hand side, or a list of the coefficients $[a, b, c]$.
 - Optionally, x the independent variable (by default x).
 - Optionally, y , the dependent variable (by default y). This option should not be used if the first argument is a list of coefficients.
- `kovacicsols(kode <x,y>)` returns a Liouvillian solution of equation (6.12). This can be a list or an expression. An empty list means that there are no Liouvillian solutions to the equation. A non-empty list will contain one or two independent solutions to the differential equation. If the list contains two solutions y_1 and y_2 , the general solution will be

$$y = C_1 y_1 + C_2 y_2$$

where $C_1, C_2 \in \mathbb{R}$ are arbitrary constants. However, for some equations only one solution y_1 is returned, in which case the other solution can be obtained as (using reduction of order):

$$y_2 = y_1 \int y_1^{-2}. \quad (6.13)$$

If `kovacicsols` returns an expression, it will give the solution of the differential equation implicitly. In that case the return value is a polynomial P of order $n \in \{4, 6, 12\}$ in the variable `omega_` (denoted here by ω) with rational coefficients r_k , $k = 0, 1, 2, \dots, n$. If $P(\omega_0) = 0$ for some ω_0 , then $y = \exp\left(\int \omega_0\right)$ is a solution to the differential equation.

Examples.

- Find the general solution to:

$$y'' = \left(\frac{1}{x} - \frac{3}{16x^2}\right) y.$$

Input:

```
kovacicsols(y'',y*(1/x-3/16x^2))
```

Output:

$$\left[x^{\frac{1}{4}}e^{2\sqrt{x}}, x^{\frac{1}{4}}e^{-2\sqrt{x}}\right]$$

Therefore, the general solution is $y = C_1 x^{1/4} e^{2\sqrt{x}} + C_2 x^{1/4} e^{-2\sqrt{x}}$.

- Solve:

$$x''(t) + \frac{3(t^2 - t + 1)}{16(t-1)^2 t^2} x(t) = 0.$$

Input:

```
kovacicsols(x'',3*(t^2-t+1)/(16*(t-1)^2*t^2)*x,t,x)
```

Output:

$$\left[\left(-t(t-1)\left(2\sqrt{t^2-t}+2t-1\right)\right)^{\frac{1}{4}}, \left(t(t-1)\left(2\sqrt{t^2-t}-2t+1\right)\right)^{\frac{1}{4}}\right]$$

so the general solution is, for $C_1, C_2 \in \mathbb{R}$,

$$x(t) = C_1 \sqrt[4]{t(t-1)(1-2t-2\sqrt{t^2-t})} + C_2 \sqrt[4]{t(t-1)(1-2t+2\sqrt{t^2-t})}.$$

- Find a particular solution to

$$y'' = \frac{4x^6 - 8x^5 + 12x^4 + 4x^3 + 7x^2 - 20x + 4}{4x^4} y.$$

Input:

```
r:=(4x^6-8x^5+12x^4+4x^3+7x^2-20x+4)/(4x^4)
kovacicsols(y'',r*y)
```

Output:

$$\left[\frac{(-1+x^2) e^{\frac{1}{2}(x^3-2x^2-2)}}{x\sqrt{x}} \right]$$

Hence $y = (x^2 - 1)x^{-3/2} e^{\frac{x^3-2x^2-2}{2x}}$ is a solution to the given equation.

- Solve

$$y'' + y' = \frac{6y}{x^2}.$$

Input:

```
kovacicsols(y'',y'=6y/x^2)
```

Output:

$$\left[\frac{(12+6x+x^2) e^{-x}}{x^2} \right]$$

- Solve the Titchmarsh equation

$$y'' + (19 - x^2)y = 0$$

Input:

```
kovacicsols(y'',(19-x^2)*y=0,x,y)
```

Output:

$$\left[\left(\frac{945}{16}x - \frac{315}{2}x^3 + \frac{189}{2}x^5 - 18x^7 + x^9 \right) e^{-\frac{x^2}{2}} \right]$$

This is only one, particular solution.

- Find the general solution of Halm's equation

$$(1+x^2)^2 y''(x) + 3y(x) = 0$$

Input:

```
sol:=kovacicsols((1+x^2)^2*y''+3y=0,x,y)
```

Output:

$$\left[\frac{-1+x^2}{\sqrt{x^2+1}} \right]$$

The other basic solution is obtained by using (6.13).

Input:

```
y1:=sol[0]; y2:=normal(y1*int(y1^-2,x))
```

Output:

$$\frac{-1+x^2}{\sqrt{x^2+1}}, -\frac{x\sqrt{x^2+1}}{x^2+1}$$

Therefore, $y = C_1 \frac{x^2-1}{\sqrt{x^2+1}} + C_2 \frac{x}{\sqrt{x^2+1}}$, where $C_1, C_2 \in \mathbb{R}$.

- Find the general solution of the non-homogeneous equation

$$y'' - \frac{27y}{36(x-1)^2} = x + 4.$$

First you need to find the general solution to the corresponding homogeneous equation $y_h'' - \frac{27y_h}{36(x-1)^2} = 0$.

Input:

```
sols:=kovaciccols(y'',-y*27/(36*(x-1)^2),x,y)
```

Output:

$$\left[\frac{-2x+x^2}{\sqrt{x-1}} \right]$$

Call this solution y_1 and find the other basic independent solution by using (6.13).

Input:

```
y1:=sols[0]:: y2:=y1*int(1/y1^2,x)
```

Output:

$$-\frac{\sqrt{x-1}}{2x-2}$$

So the general solution of the homogeneous equation is

$$y_h = C_1 y_1 + C_2 y_2 = \frac{C_1 (x^2 - 2x) + C_2}{\sqrt{x-1}}, \quad C_1, C_2 \in \mathbb{R}.$$

A particular solution y_p of the non-homogeneous equation can be obtained by variation of parameters as

$$y_p = -y_1 \int \frac{y_2 f(x)}{W} dx + y_2 \int \frac{y_1 f(x)}{W} dx,$$

where $f(x) = x + 4$ and W is the Wronskian of y_1 and y_2 , i.e.

$$W = y_1 y_2' - y_2 y_1' \neq 0.$$

Input:

```
W:=y1*y2'-y2*y1':; f:=x+4::
yp:=normal(-y1*int(y2*f/W,x)+y2*int(y1*f/W,x))
```

Output:

$$\frac{4x^3 + 72x^2 - 156x + 80}{21}$$

Hence $y_p = \frac{1}{21}(4x^3 + 72x^2 - 156x + 80)$. Now $y = y_p + y_h$. You can check that it is indeed the general solution of the given equation.

Input:

```
purge(C1,C2); ysol:=yp+C1*y1+C2*y2;;
normal(diff(ysol,x,2)-27/(36*(x-1)^2)*ysol)==f
```

Output:

true

- Solve the equation:

$$y'' = \left(\frac{3}{16x(x-1)} - \frac{2}{9(x-1)^2} - \frac{3}{16x^2} \right) y.$$

from the original Kovacic's paper:

Input:

```
r:=-3/(16x^2)-2/(9*(x-1)^2)+3/(16x*(x-1));
kovaciccsols(y',r*y)
```

Output:

$$\begin{aligned} & -\omega_4 x^4 (x-1)^4 + \omega_3 x^3 (x-1)^3 (7x-3)/3 - \omega_2 x^2 (x-1)^2 (48x^2 - 41x + 9)/24 + \\ & -\omega_1 x (x-1) (320x^3 - 409x^2 + 180x - 27)/432 + \\ & (-2048x^4 + 3484x^3 - 2313x^2 + 702x - 81)/20736 \end{aligned}$$

The solution is $y = \exp\left(\int \omega_0\right)$, where ω_0 is a zero of the above expression, thus being a root of a fourth-order polynomial in ω . In similar cases you can try the Ferrari method to obtain ω_0 .

- Solve the equation

$$48t(t+1)(5t-4)y'' + 8(25t+16)(t-2)y' - (5t+68)y = 0.$$

Input:

```
kovaciccsols([48t*(t+1)*(5t-4),8*(25t+16)*(t-2),-(5t+68)],t)
```

Output:

$$\frac{1}{20736}\omega_4(135t^4 - 616t^3 - 144t^2 + 3072t - 4096) - \frac{1}{54}\omega_3 t(t+1)(23t^2 - 92t + 128) - \frac{1}{24}\omega_2 t^2(t+1) + \omega_1 t(t+1)(5t-4) - (5t+68)$$

6.58 The Z-transform

6.58.1 The Z-transform of a sequence: `ztrans`

The Z-transform of a sequence $a_0, a_1, \dots, a_n, \dots$ is the function

$$f(z) = \sum_{n=0}^{\infty} \frac{a_n}{z^n}.$$

For example, the Z-transform of the sequence

$$0, 1, 2, 3, \dots$$

is

$$f(z) = 0 + 1/z + 2/z^2 + 3/z^3 + \dots$$

which has closed form

$$f(z) = z/(z-1)^2.$$

The `ztrans` command finds the Z-transform of a sequence.

- **`ztrans`** takes one mandatory and two optional arguments:
 - a_x , a formula with a variable for the general term of a sequence.
 - Optionally, x , the variable (by default `x`).
 - Optionally, z , a variable to be used by the resulting function.

`ztrans($a_x \langle x, z \rangle >$)` returns the Z-transform of the sequence.

Examples.

- *Input:*

`ztrans(x)`

Output:

$$x/(x^2 - 2 * x + 1)$$

- *Input:*

`ztrans(n,n,z)`

Output:

$$\frac{z}{z^2 - 2z + 1}$$

- *Input:*

`ztrans(1)`

Output:

$$\frac{x}{x-1}$$

since

$$\sum_{n=0}^{\infty} 1/x^n = 1/(1 - 1/x) = x/(x-1).$$

You also have

Input:

`ztrans(1,n,z)`

Output:

$$\frac{z}{z-1}$$

Note that differentiating both sides of

$$\sum_{n=0}^{\infty} 1/z^n = z/(z-1)$$

gives you

$$\sum_{n=0}^{\infty} n/z^{n-1} = 1/(z-1)^2$$

and so, multiplying both sides by z ,

$$\sum_{n=0}^{\infty} n/z^n = z/(z-1)^2 = z/(z^2 - 2z + 1)$$

as indicated above.

6.58.2 The inverse Z-transform of a rational function: `invztrans`

The inverse Z-transform of a rational expression is a formula for the general term of a sequence with the given rational expression as its Z-transform. The `invztrans` command finds the inverse Z-transform of a rational expression.

- `invztrans` command takes one mandatory and two optional arguments:
 - `rat`, a rational expression.
 - Optionally, x the variable (by default `x`).
 - Optionally, n , a variable to be used by the result (by default x).
- `ztrans(rat <x,n>)` returns the inverse Z-transform of rat .

Examples.

- *Input:*

```
invztrans(x/(x-1))
```

Output:

1

(since `ztrans(1)=x/(x-1)`)

• *Input:*

```
invztrans(z/(z-1),z,n)
```

Output:

1

• *Input:*

```
invztrans(x/(x-1)^2)
```

Output:

x

• *Input:*

```
invztrans(z/(z-1)^2,z,n)
```

Output:

n

6.59 Other functions

6.59.1 Replacing small values by 0: `epsilon2zero`

The `epsilon2zero` command replaces small values by 0.

- `epsilon2zero` takes one argument:
expr, an expression in `x`.
- `epsilon2zero(expr)` returns *expr* where any values of modulus less than `epsilon` (see Section 3.5.7 p.73, item 9, by default `epsilon=1e-10`) are replaced by zero. The expression is not evaluated.

Examples.

• *Input:*

```
epsilon2zero(1e-13+x)
```

Output (with `epsilon=1e-10`):

$0 + x$

- *Input:*

```
epsilon2zero((1e-13+x)*100000)
```

Output (with `epsilon=1e-10`):

$$100000(0 + x)$$

- *Input:*

```
epsilon2zero(0.001+x)
```

Output (with `epsilon=0.0001`):

$$0.001 + x$$

6.59.2 List of variables: `lname` `indets`

The `lname` command finds the symbolic variable names used in an expression. `indets` is a synonym for `lname`.

- `lname` takes one argument:
expr, an expression.
- `lname(expr)` returns the list of the symbolic variable names used in *expr*.

Examples.

- *Input:*

```
lname(x*y*sin(x))
```

Output:

$$[x, y]$$

- *Input:*

```
a:=2; assume(b>0); assume(c=3);  
lname(a*x^2+b*x+c)
```

Output:

$$[x, b, c]$$

6.59.3 List of variables and of expressions: `lvar`

The `lvar` command finds the variables and non-rational that make up an expression.

- `lvar` takes one argument: *expr*, an expression.
- `lvar(expr)` returns a list of variable names and non-rational expressions such that *expr* its argument is a rational function with respect to the variables and expressions of the list.

Examples.

- *Input:*

```
lvar(x*y*sin(x)^2)
```

Output:

```
[x, y, sin x]
```

- *Input:*

```
lvar(x*y*sin(x)^2+ln(x)*cos(y))
```

Output:

```
[x, y, sin x, ln x, cos y]
```

- *Input:*

```
lvar(y+x*sqrt(z)+y*sin(x))
```

Output:

```
[y, x, sqrt(z), sin x]
```

6.59.4 List of variables of an algebraic expressions: `algvar`

The `algvar` command finds the symbolic variable names in an expression, but orders them.

- `algvar` takes one argument:
expr, an expression.
- `algvar(expr)` returns the list of the symbolic variable names used in *expr*, ordered by the algebraic extensions required to build *expr*.

Examples.

- *Input:*

```
algvar(y+x*sqrt(z))
```

Output:

```
[[y, x], [z]]
```

- *Input:*

```
algvar(y*sqrt(x)*sqrt(z))
```

Output:

$$\begin{bmatrix} y \\ z \\ x \end{bmatrix}$$

- *Input:*

```
algvar(y*sqrt(x*z))
```

Output:

$$[[y], [x, z]]$$

- *Input:*

```
algvar(y+x*sqrt(z)+y*sin(x))
```

Output:

$$[[y, x, \sin x], [z]]$$

6.59.5 Testing if a variable is in an expression: `has`

The `has` command determines whether or not an expression contains a variable.

- `has` takes two arguments:
 - `expr`, an expression.
 - `x`, the name of a variable.
- `has(expr, x)` returns 0 if `expr` doesn't contain `x`, otherwise it returns an integer giving the position of `x` on `lname(expr)` (where the position starts at 1).

Examples.

- *Input:*

```
has(x*y*sin(x), y)
```

Output:

2

- *Input:*

```
has(x*y*sin(x), z)
```

Output:

0

6.59.6 Numeric evaluation: `evalf`

The `evalf` command finds floating point approximations to the numbers in an expression or a matrix (see also Section 6.8.1 p.171).

- `evalf` takes one mandatory and one optional argument:
 - *expr*, an expression or a matrix.
 - Optionally, *n*, a positive integer representing the significant digits (by default `DIGITS`, which itself has a default value of 12; see Section 3.5.1 p.70).
- `evalf(expr <n>)` returns *expr* with all the numbers replaced by floating point approximations to *n* digits.

Examples.

- *Input:*

```
evalf(sqrt(2))
```

Output:

```
1.41421356237
```

- *Input:*

```
evalf(sqrt(2), 20)
```

Output:

```
1.4142135623730950488
```

- *Input:*

```
evalf([[1, sqrt(2)], [0, 1]])
```

Output:

$$\begin{bmatrix} 1.0 & 1.41421356237 \\ 0.0 & 1.0 \end{bmatrix}$$

6.59.7 Rational approximation: `float2rational` `exact`

Floating point numbers are considered approximations, while integers and rational numbers are considered exact. The `float2rational` command finds a rational approximation to a floating point number.

`exact` is a synonym for `float2rational`.

- `float2rational` takes one argument:
 - *expr*, an expression.
- `float2rational(expr)` returns *expr* with all the floating point numbers in *expr* replaced by rational numbers; any floating point number *x* is replaced by a rational *r* with $|r - x| < \epsilon$, where ϵ is given by `epsilon` in the `cas` configuration (see Section 3.5.7 p.73, item 9).

Examples.

- *Input:*

```
float2rational(1.5)
```

Output:

$$\frac{3}{2}$$

- *Input:*

```
float2rational(1.414)
```

Output:

$$\frac{707}{500}$$

- *Input:*

```
float2rational(0.156381102937*2)
```

Output:

$$\frac{5144}{16447}$$

- *Input:*

```
float2rational(1.41421356237)
```

Output:

$$\frac{114243}{80782}$$

Input:

```
float2rational(1.41421356237^2)
```

Output:

$$2$$

6.60 The day of the week: dayofweek

The `dayofweek` command finds the day of the week for any date after 15 October, 1582.

- `dayofweek` takes as arguments three arguments:
 - d , an integer representing the day of the month.
 - m , an integer representing the month.
 - y , an integer representing the year.

The date represented should be after 15 October 1582.

- `dayofweek(d, m, y)` returns an integer from 0 to 6; 0 representing Sunday, 1 representing Monday, etc.

Examples.

- *Input:*

`dayofweek(1, 10, 2014)`

Output:

3

This means that 1 October, 2014 was a Wednesday.

- *Input:*

`dayofweek(15, 10, 1582)`

Output:

5

This indicates that 15 October 1582 was on a Friday.

The Gregorian calendar, the calendar used by most of the world, was introduced on 15 October 1582. Before that, the Julian calendar was used, which had a leap year every four years and so used years with an average of 365.25, which is slightly off from the actual value of about 365.242 days. To deal with this, the Gregorian calendar was introduced, where a leap year is a year which is divisible by 4, but not divisible by 100 unless it is also divisible by 400. This gives an average length of year that is accurate to within 1 day every 3000 years.

Many countries switched from the Julian calendar to the Gregorian calendar after 4 October 1582 in the Julian calendar, and the next day was 15 October 1582.

Chapter 7

Metric properties of curves

7.1 The center of curvature

Let Γ be a curve in space parameterized by a continuously differentiable function, and M_0 be a point on the curve. The curve will have an arclength parameterization; namely, it can be parameterized by a function $M(s)$, where $M(0) = M_0$ and $|s|$ is the length of the curve from M_0 to $M(s)$, in the direction of the curve if $s > 0$ and the opposite direction if $s < 0$.

For such a Γ , the vector $T(s) = M'(s)$ will be the unit tangent to the curve at $M(s)$, and $N(s) = T'(s)$ will be perpendicular to the tangent. The circle through $M(s)$ with center at $M(s) + N(s)$ is called the *osculating circle* to Γ at $M(s)$. Informally, the osculating circle is the circle through $M(s)$ which most closely approximates Γ . The set of all centers of curvature is another curve, called the *evolute* of Γ .

The radius of the osculating circle is $|N(s)|$ and is called the *radius of curvature* of Γ at $M(s)$. The reciprocal of this is called the *curvature* of Γ at $M(s)$.

7.2 Computing the curvature and related values: curvature osculating_circle evolute

A curve can be described in `Xcas` with a parametrization or with a curve object. Various curve objects are described in chapters 13 and 14. The commands in this section can work with curves described either way. You can get the equation of a curve object with the `equation` command (see Section 13.13.7 p.939).

The `curvature` command finds the curvature of a curve. The curve can be given as an object or by a parameterization.

To find the curvature from a parameterization:

- `curvature` takes two mandatory arguments and one optional argument:
 - C , a curve.
 - t , the parameter of the curve.

- Optionally, t_0 , a value of the parameter.
- `curvature($C, t \langle, t_0 \rangle$)` returns the curvature of the curve; if t_0 is given, the curvature is given at the point it specifies, otherwise the curvature is given as a function of the parameter.

To find the curvature from a curve object:

- `curvature` takes two arguments:
 - C , a curve.
 - p , a point on the curve.
- `curvature(C, p)` returns the curvature of C at the point p .

Examples.

- *Input:*

```
trigsimplify(curvature([5*cos(t),5*sin(t)],t))
```

Output:

$$\frac{1}{5}$$

- *Input:*

```
curvature([2*cos(t),3*sin(t)],t)
```

Output:

$$\frac{(6 \cos^2 t + 6 \sin^2 t) \sqrt{9 \cos^2 t + 4 \sin^2 t}}{(9 \cos^2 t + 4 \sin^2 t)^2}$$

- *Input:*

```
curvature([2*cos(t),3*sin(t)],t,pi/2)
```

Output:

$$\frac{3}{4}$$

- *Input:*

```
curvature(plot(x^2),point(1,1))
```

(see Section 8.5 p.673 and Section 13.6.2 p.886.) *Output:*

$$\frac{2}{25}\sqrt{5}$$

The `osculating_circle` command finds and draws the osculating circle of a curve.

To find the osculating circle from a parameterization:

7.2. COMPUTING THE CURVATURE AND RELATED VALUES: CURVATURE OSCULATING_CIRCLE EVOLU

- `osculating_circle` takes three arguments:
 - C , a curve.
 - t , the parameter of the curve.
 - t_0 , a value of the parameter.
- `osculating_circle(C, t, t_0)` draws and returns the osculating circle of the curve at the point specified by t_0 .

To find the osculating circle from a curve object:

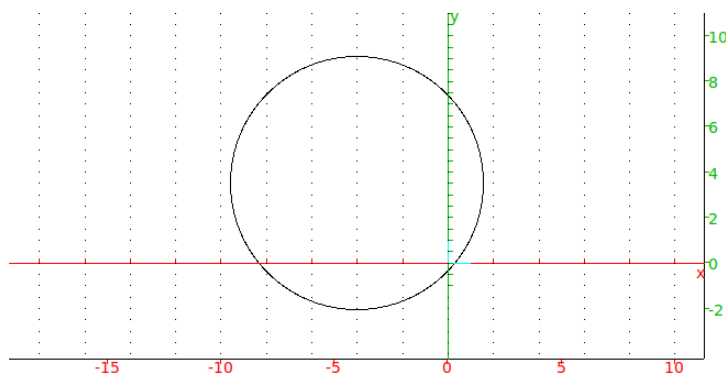
- `osculating_circle` takes two arguments:
 - C , a curve.
 - p , a point on the curve.
- `osculating_circle(C, p)` draws and returns the osculating circle of C at the point p .

Examples.

- *Input:*

```
osculating_circle(plot(x^2),point(1,1))
```

Output:



- *Input:*

```
equation(osculating_circle(plot(x^2),point(1,1)))
```

Output:

$$(x + 4)^2 + \left(y - \frac{7}{2}\right)^2 = \frac{125}{4}$$

- *Input:*

```
equation(osculating_circle([t^2,t^3],t,1))
```

Output:

$$\left(x + \frac{11}{2}\right)^2 + \left(y - \frac{16}{3}\right)^2 = \frac{2197}{36}$$

The **evolute** command finds and draws the evolute of a curve. To find the evolute from a parameterization:

- **evolute** takes two arguments:
 - C , a curve.
 - t , the parameter of the curve.
- **evolute**(C, t) draws and returns the evolute of the curve.

To find the evolute from a curve object:

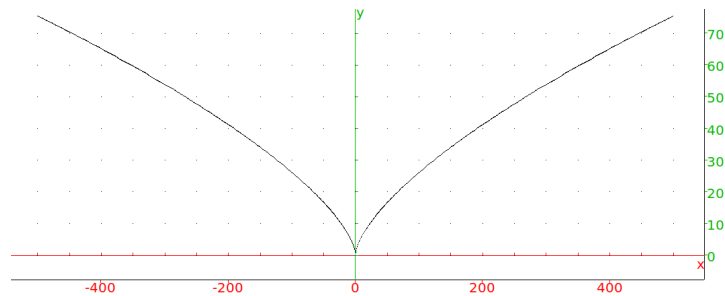
- **evolute** takes one argument:
 - C , a curve.
- **evolute**(C) draws and returns the evolute of C .

Examples.

- *Input:*

```
evolute(plot(x^2))
```

Output:



- *Input:*

```
equation(evolute(plot(x^2)))
```

Output:

$$27x^2 - 16y^3 + 24y^2 - 12y + 2 = 0$$

- *Input:*

```
equation(evolute([t^2,t],t))
```

Output:

$$16x^3 - 24x^2 + 12x - 27y^2 - 2 = 0$$

Chapter 8

Graphs

8.1 Generalities

Most graph instructions take expressions as arguments. A few exceptions (mostly Maple-compatibility instructions) also accept functions. Some optional arguments, like `color`, `thickness`, can be used as optional attributes in all graphic instructions. They are described below.

If a graph depends on a user-defined function, you may want to define the function when the parameter is a formal variable. For this, it can be useful to test the type of the parameter while the function is being defined. (See Chapter Section 12 p.835 for information about programming in Xcas.)

For example, suppose `f` and `g` are defined by:

```
f(x) := {
  if (type(x) != DOM_FLOAT) return 'f'(x);
  while(x > 0){ x--;}
  return x;
}
```

and

```
g(x) := {
  while(x > 0){ x--;}
  return x;
};;
```

Graphing these (see Section 8.4.1 p.667):

Input:

```
F := plotfunc(f(x))
G := plotfunc(g(x))
```

they will both produce the same graph. However, the graphic `G` won't be reusable. Entering:

Input:

F

reproduces the graph, but entering:

Input:

G

produces the error:

Output:

```
"Unable to eval test in loop:  x>0.0
Error:  Bad Argument Value Error:
Bad Argument Value"
```

Internally, **F** and **G** contain the formal expressions $f(x)$ and $g(x)$, respectively. When **Xcas** tries to evaluate **F** and **G**, x has no value and so the test $x > 0$ produces an error in $g(x)$, but the line `if (type(x)!=DOM_FLOAT) return 'f'(x);` avoids this problem in $f(x)$.

8.2 The graphic screen

A graphic screen, either two- or three-dimensional as appropriate, automatically opens in response to a graphic command. Alternatively, you can open a graphic screen with its own command line with keystrokes; **Alt-g** for a two-dimensional screen and **Alt-h** for a three-dimensional screen. The graphic screen will have an array of buttons at the top right.

- There will be red arrows for moving the image in the x direction.
- There will be green arrows for moving the image in the y direction.
- There will be blue arrows for zooming in and out in a two-dimensional screen, and moving the image in the z direction in a three-dimensional screen.
- There will be **in** and **out** buttons for zooming in and out.
- There will be a **_|_** button to orthonormalize the graphic.
- There will be a **▶|** button to start and stop animations.
- There will be an **auto** button to do automatic scaling.
- There will be a **cfg** button which will bring up a configuration screen (see Section 3.5.7 p.73).
- There will be an **M** button which brings up a menu. The menu has submenus:
 - **View** which has entries which do the same as the buttons.
 - **Trace** for working with traces.
 - **Animation** for working with animations.
 - **3-d** for working with three-dimensional graphics.
 - **Export/Print** to export and print the graphic.

The image can also be moved in the screen by clicking and dragging with the mouse. Scrolling with the mouse will also zoom the images.

8.3 Graph and geometric objects attributes

There are two kinds of attributes for graphs and geometric objects: global attributes of a graphic scene and individual attributes.

8.3.1 Individual attributes

Graphic attributes are optional arguments of the form `display=value`. They must be given as the last argument of a graphic instruction. Attributes are ordered in several categories: color, point shape, point width, line style, line thickness, legend value, position and presence. In addition, surfaces may be filled or not, 3-d surfaces may be filled with a texture, 3-d objects may also have properties with respect to the light. Attributes of different categories may be combined with +, e.g.

```
plotfunc( $x^2 + y^2$ , [x,y], display=red+line_width_3+filled)
```

The graphic attributes are:

- Colors, set with `display=value` or `color=value`. The values can be:
 - black, white, red, blue, green, magenta, cyan, yellow,
 - a numeric value between 0 and 255,
 - a numeric value between 256 and $256+7*16+14$ for a color of the rainbow,
 - any other numeric value smaller than 65535, the rendering is not guaranteed to be portable.
- Point shapes, set with `display=value`. The values can be:
 - rhombus_point plus_point
 - square_point cross_point
 - triangle_point
 - star_point
 - point_point
 - invisible_point
 - Point width, set with `display=value`. The values can be: `point_width_n` where n is an integer between 1 and 7.
 - Line thickness, set with `thickness=n` or `display=line_width_n` where n is an integer between 1 and 7.
 - Line shape, set with `display=value`. The values can be:
 - * dash_line
 - * solid_line
 - * dashdot_line
 - * dashdotdot_line
 - * cap_flat_line
 - * cap_square_line
 - * cap_round_line

- Legend, the text is set with `legend="legendname"`; the position is set with `display=value`, where the values can be:

- * `quadrant1`
- * `quadrant2`
- * `quadrant3`
- * `quadrant4`

and correspond to the position of the legend of the object (using the trigonometric plane conventions). The legend is not displayed if the attribute `display=hidden_name` is added.

- `display=filled` specifies that surfaces will be filled,
- `gl_texture="picture_filename"` is used to fill a surface with a texture. Cf. the interface manual for a more complete description and for `gl_material=` options.

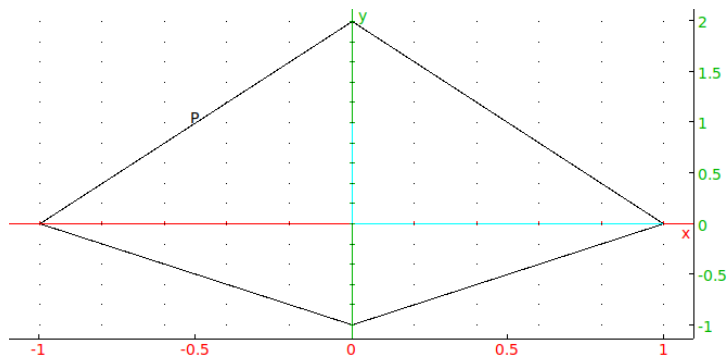
Examples.

(See Section 13.10.3 p.920, Section 13.6.2 p.886, Section 13.3.3 p.881 and Section 13.7.3 p.900 for information on the commands used.)

- *Input:*

```
polygon(-1,-i,1,2*i,legend="P")
```

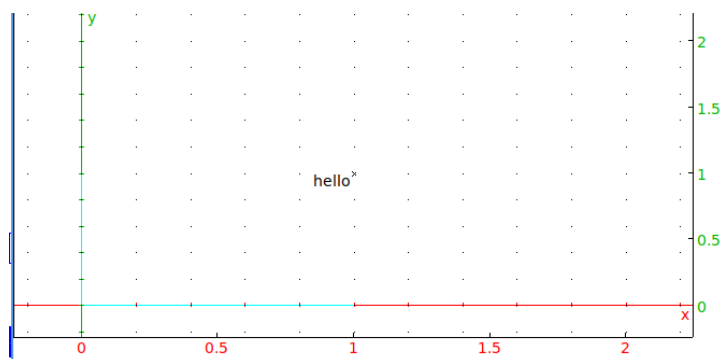
Output:



- *Input:*

```
point(1+i,legend="hello")
```

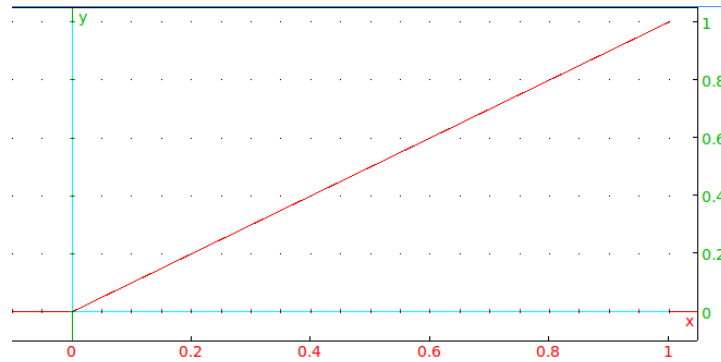
Output:



– *Input:*

```
color(segment(0,1+i),red)
```

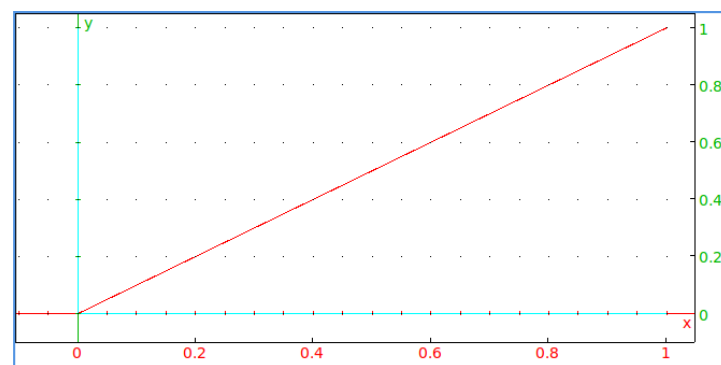
Output:



– *Input:*

```
segment(0,1+i,color=red)
```

Output:



8.3.2 Global attributes

These attributes are shared by all objects of the same scene

- `title="titlename"` sets the title.
- `labels=["xname","yname","zname"]` sets names of the x, y, z axes.
- `gl_x_axis_name="xname", gl_y_axis_name="yname", gl_z_axis_name="zname"` sets the names of the axes individually.
- `legend=["xunit","yunit","zunit"]` sets units for the axes.
- `gl_x_axis_unit="xunit", gl_y_axis_unit="yunit", gl_z_axis_unit="zunit"` sets units for the axes individually.
- `axes=true` or `axes=false` shows or hides the axis.
- `gl_texture="filename"` sets the background image to `"filename"`.
- `gl_x=xmin..xmax, gl_y=ymin..ymax, gl_z=zmin..zmax` sets the graphic configuration (do not use for interactive scenes)

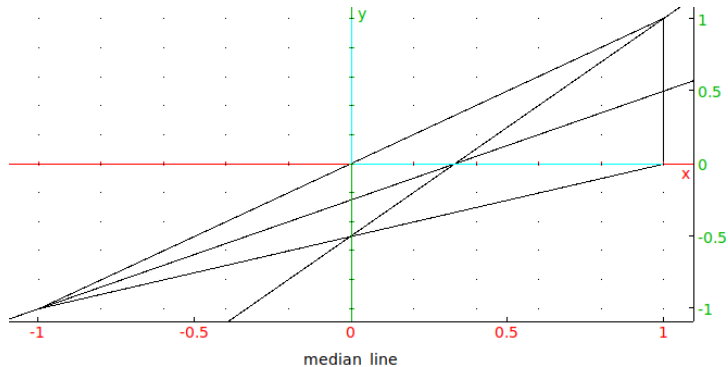
- `gl_xtick=xmark, gl_ytick=ymark, gl_ztick=zmark` sets the tick marks for the axes.
- `gl_shownames=true` or `gl_shownames=false` shows or hides objects names
- `gl_rotation=[x,y,z]`: defines the rotation axis for the animation rotation of 3-d scenes.
- `gl_quaternion=[x,y,z,t]`: defines the quaternion for the visualization in 3-d scenes (do not use for interactive scenes)
- a few other OpenGL light configuration options are available but not described here.

Examples.

- *Input:*

```
title="median_line";triangle(-1-i,1,1+i);median_line(-1-i,1,1+i);median.
```

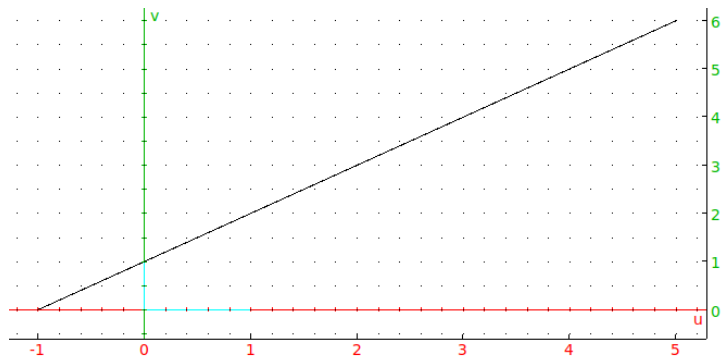
Output:



- *Input:*

```
labels=["u", "v"];plotfunc(u+1,u)
```

Output:



8.4 Graph of a function: plotfunc funcplot DrawFunc Graph

8.4.1 2-d graph

The `plotfunc` command draws the graph of a function. `funcplot` is a synonym for `plotfunc`.

`plotfunc` can draw the graph of a one-variable function or a two-variable function; this section will discuss one-variable functions and the next section will discuss two-variable functions.

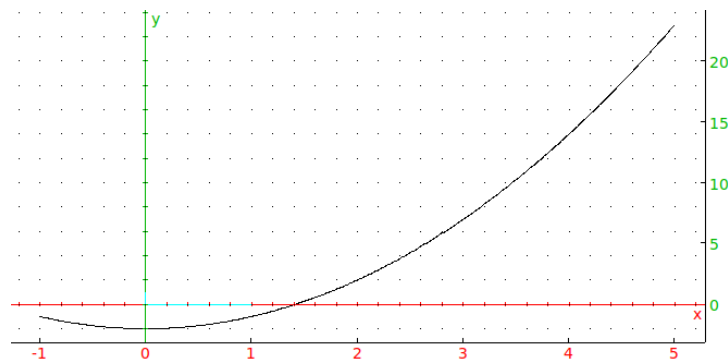
- `plotfunc` takes one mandatory argument and two optional arguments:
 - * *expr*, an expression defining a function.
 - * Optionally, *var*, the variable name (by default `x`) possibly with bounds. If the variable is given as *var=a..b*, the graph will be drawn from *a* to *b*, otherwise it will be graphed over the default interval (see Section 3.5.8 p.76).
 - * Optionally, *opt*, which can be `xstep=n` to specify the discretization step or `nstep=n` to specify the number of points used to graph.
- `plotfunc(expr,var<opt>)` draws the graph.

Examples.

- *Input:*

```
plotfunc(x^2-2)
```

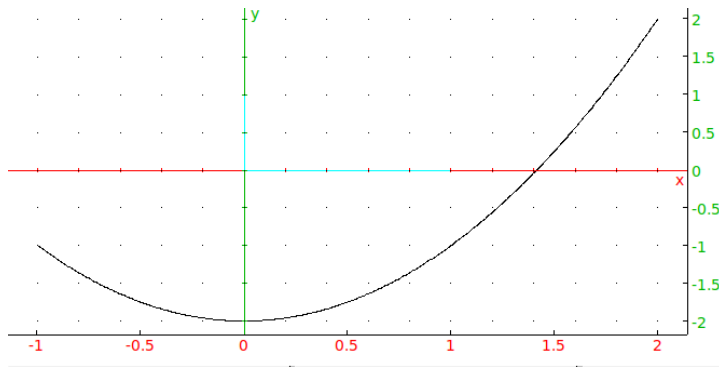
Output:



- *Input:*

```
plotfunc(a^2-2,a=-1..2)
```

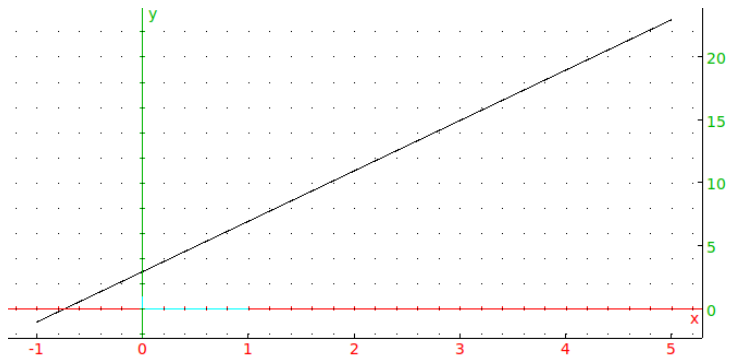
Output:



– *Input:*

```
plotfunc(x^2-2,x,xstep=5)
```

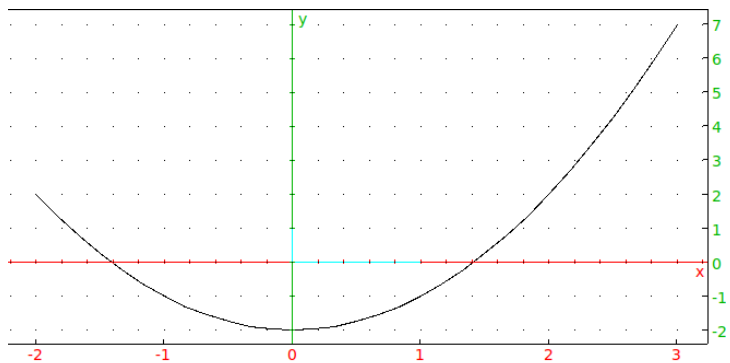
Output:



– *Input:*

```
plotfunc(x^2-2,x=-2..3,nstep=30)
```

Output:



8.4.2 3-d graph

Two variable functions

The `plotfunc` can draw the graphs of two-variable function.

- `plotfunc` takes two mandatory argument and two optional arguments:

- * *expr*, an expression defining a function of two variables or a list of such expressions.
 - * *vars*, a list of the variable names, possibly with bounds. If the variable is given as *var=a..b*, the graph will be drawn for that range of that variable, otherwise it will be graphed over the default interval (see Section 3.5.8 p.76).
 - * Optionally, *xstep*, which can be **xstep=n** to specify the discretization step in the *x* direction.
 - * Optionally, *ystep*, which can be **ystep=m** to specify the discretization step in the *y* direction.
 - * Instead of *xstep* and *ystep*, you could use the option **nstep=n** to specify the number of points used to graph.
- `plotfunc(expr,vars <xstep,ystep>)` draws the graph.

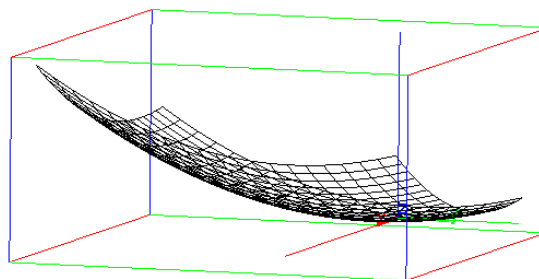
Examples.

– *Input:*

```
plotfunc(x^2+y^2,[x,y])
```

Output:

mouse plan 0.957x+0.289y+0.0287z=2.16



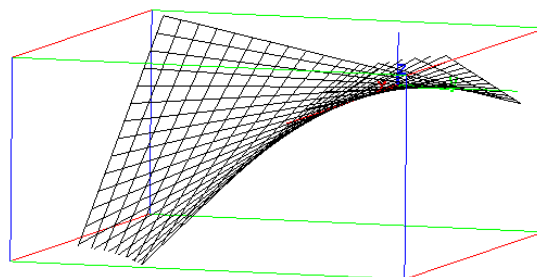
click k: kill 3d view

– *Input:*

```
plotfunc(x*y,[x,y])
```

Output:

mouse plan 0.954x+0.288y+0.0834z=1.09

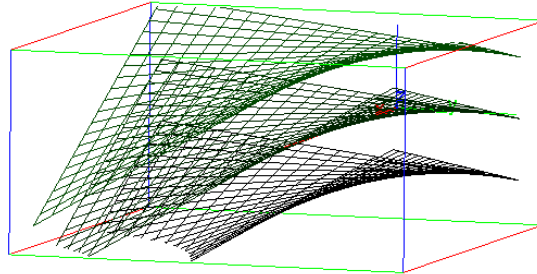


– *Input:*

```
plotfunc([x*y-10,x*y,x*y+10],[x,y])
```

Output:

mouse plan 0.956x+0.288y+0.0499z=1.34

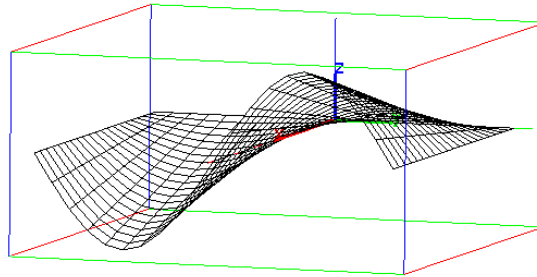


– *Input:*

```
plotfunc(x*sin(y),[x=0..2,y=-pi..pi])
```

Output:

mouse plan 0.984x+0.116y+0.136z=0.93



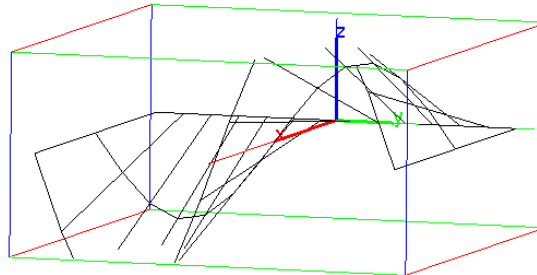
– As an example where you specify the x and y discretization step with `xstep` and `ystep`:

Input:

```
plotfunc(x*sin(y),[x=0..2,y=-pi..pi],xstep=1,ystep=0.5)
```

Output:

mouse plan 0.965x+0.114y+0.236z=0.912



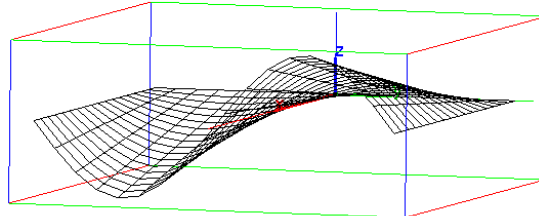
– Alternatively you can specify the number of points used for the representation of the function with `nstep` instead of `xstep` and `ystep`.

Input:

```
plotfunc(x*sin(y),[x=0..2,y=-pi..pi],nstep=300)
```

Output:

mouse plan 0.984x+0.116y+0.136z=0.93



Remarks:

- Like any 3-d scene, the viewpoint may be modified by rotation around the x axis, the y axis or the z axis, either by dragging the mouse inside the graphic window (push the mouse outside the parallelepiped used for the representation), or with the shortcuts x , X , y , Y , z and Z .
- If you want to print a graph or get a \LaTeX translation, use the graph menu
Menu►print►Print(with Latex)

3-d graph with rainbow colors

If the expression with two variables is purely imaginary, $iexpr$, then `plotfunc` will still draw the graph, but the color will depend on the height $z = expr$ resulting in a rainbow colored surface. This provides you with an easy way to find points having the same third coordinate.

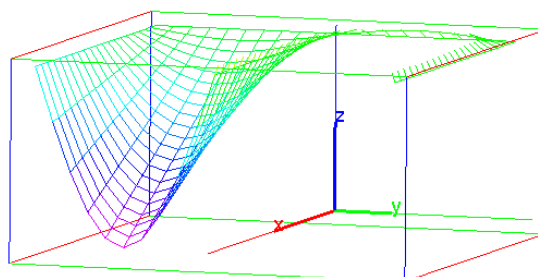
Example.

Input:

```
plotfunc(i*x*sin(y),[x=0..2,y=-pi..pi])
```

Output:

mouse plan 0.961x+0.113y+0.252z=1.15



4-d graph

If $expr$ is a complex valued expression whose real part is not identically zero on the discretization mesh, then `plotfunc` will draw the surface $z = \text{abs}(expr)$, where `arg(expr)` determines the color from the rainbow. This gives you an easy way to see the points having the same argument. Note that if the real part of $expr$ is zero on the discretization mesh, then it will look purely imaginary to `plotfunc` and will be represented with rainbow colors, as in Section 8.4.2 p.671.

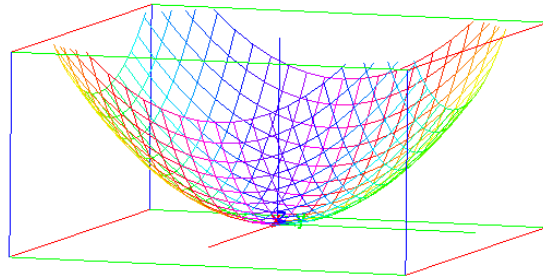
Examples.

– *Input:*

```
plotfunc((x+i*y)^2,[x,y])
```

Output:

mouse plan 0.943x+0.332y+0.0326z=2.23

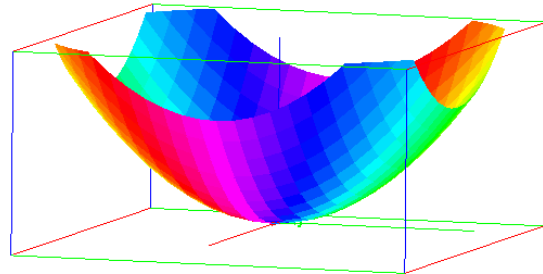


– *Input:*

```
plotfunc((x+i*y)^2,[x,y], display=filled)
```

Output:

mouse plan 0.943x+0.332y+0.0326z=2.23



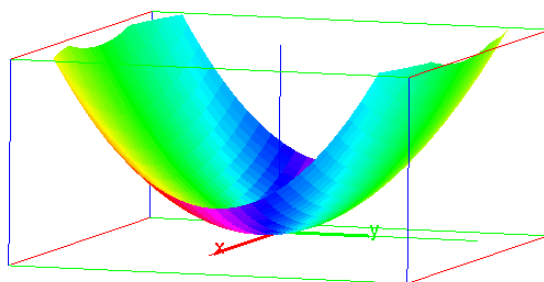
– You can specify the range of variation of x and y and the number of discretization points.

Input:

```
plotfunc((x+i*y)^2,[x=-1..1,y=-2..2],
         nstep=900,display=filled)
```

Output:

mouse plan $0.978x+0.172y+0.122z=0.231$



8.5 2d graph for Maple compatibility: plot

The `plot` command is a Maple-compatible way to draw the graph of a one-variable function.

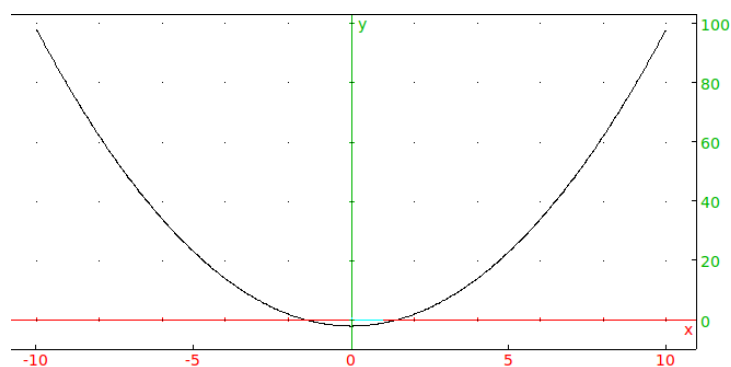
- `plot` takes one mandatory argument and two optional arguments:
 - * *func*, a function or an expression involving one variable.
 - * Optionally, *var* the name of the variable in the expression (if *func* is an expression), which can also specify a range of values *var=a..b* (by default it is *x*). If *func* is a function, the optional second argument can simply be a range *a..b* for the variable.
 - * Optionally, *opt*, which can be *xstep=n* to specify the discretization step or *nstep=n* to specify the number of points used to graph.
- `plot(expr⟨,var, opt⟩)` draws the graph.

Examples.

- *Input:*

```
plot(x^2-2,x)
```

Output:



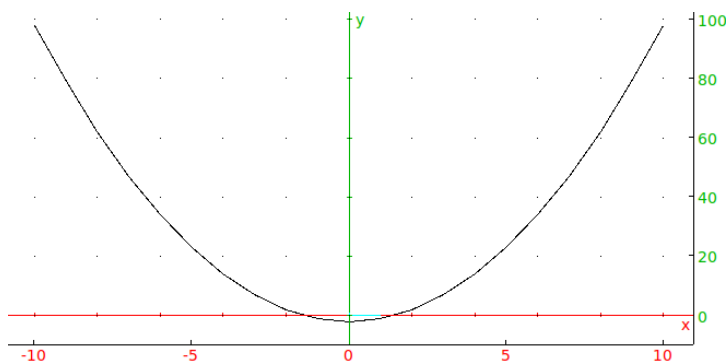
- *Input:*

```
plot(x^2-2,xstep=1)
```

or:

```
plot(x^2-2,x,xstep=1)
```

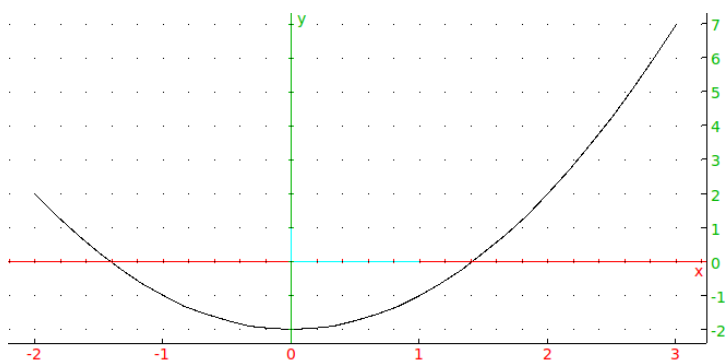
Output:



– Input:

```
plot(x^2-2,x=-2..3,nstep=30)
```

Output:



8.6 3d surfaces for Maple compatibility plot3d

The `plot3d` command is a Maple-compatible way to draw a surface. It can plot the graph of a function of two variables or a surface given by a parameterization.

To draw the graph of a function:

– `plot3d` takes three arguments:

- * *func*, a function or an expression involving two variables.
- * *x* and *y*, the names of the variable in the expression (if *func* is an expression) which can also specify a range of values for each variable.

If *func* is a function, this argument is optional, and are the ranges *a..b* for the variables.

If the ranges are not given, the default values are taken from the graph configuration (see Section 3.5.8 p.76).

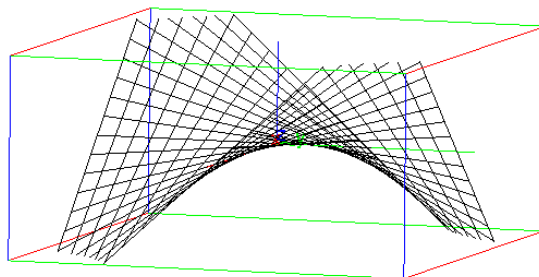
- `plot3d(func, x, y)` draws the graph.

Example.*Input:*

```
plot3d(x*y, x, y)
```

Output:

mouse plan 0.942x+0.332y+0.0454z=0



To draw a parameterized surface:

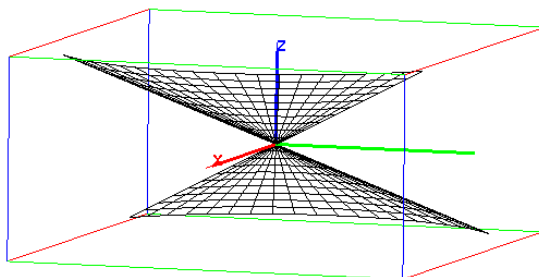
- `plot3d` takes one mandatory argument and two optional arguments:
 - * *funcs*, a list of three functions or three expressions involving two variables.
 - * *u* and *v*, the names of the variable in the expression (if *funcs* is a list of expressions), which can also specify a range of values for each variable.
 If *funcs* is a list of functions, this argument is optional, and are the ranges *a..b* for the variables.
 If the ranges are not given, the default values are taken from the graph configuration (see Section 3.5.8 p.76).
- `plot3d(funcs, u, v)` draws the surface.

Examples.*Input:*

```
plot3d([v*cos(u), v*sin(u), v], u, v)
```

Output:

mouse plan 0.897x+0.375y+0.235z=0

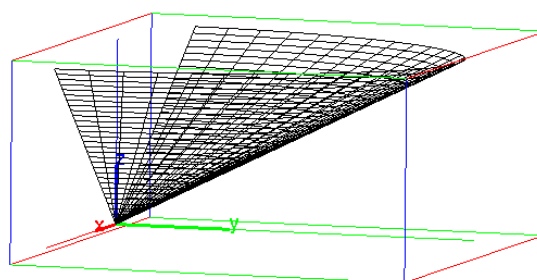


– *Input:*

```
plot3d([v*cos(u),v*sin(u),v],u=0..pi,v=0..3)
```

Output:

mouse plan 0.767x+0.514y+0.383z=1.27



8.7 Graph of a line and tangent to a graph

8.7.1 Drawing a line: line

The `line` command draws and finds lines in \mathbb{R}^2 and \mathbb{R}^3 .

For a line in \mathbb{R}^2 :

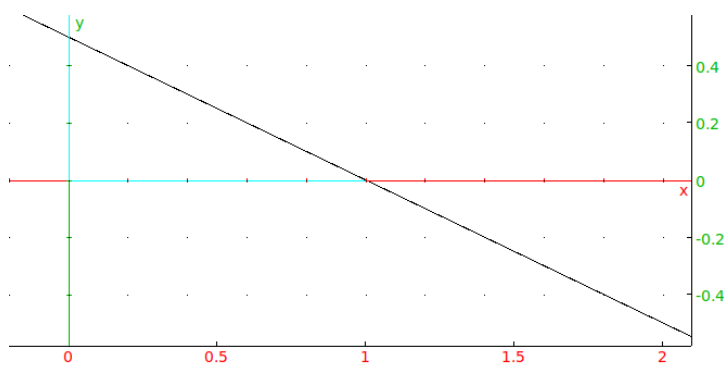
- `line` takes one argument:
`eqn`, a linear equation in the variables `x` and `y`.
- `line(eqn)` draws and returns the line given by the equation.

Examples.

– *Input:*

```
line(2*y+x-1=0)
```

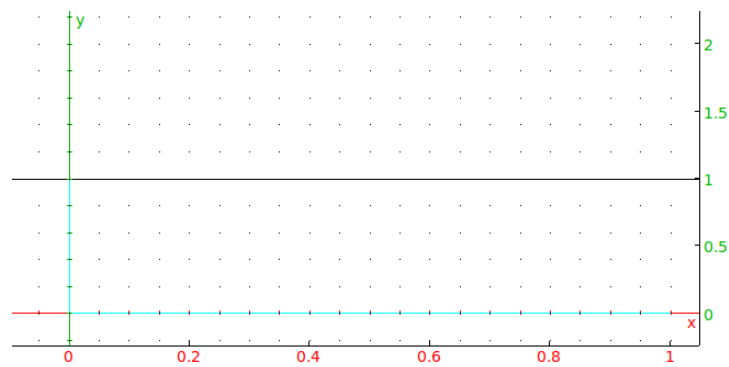
Output:



– *Input:*

```
line(y=1)
```

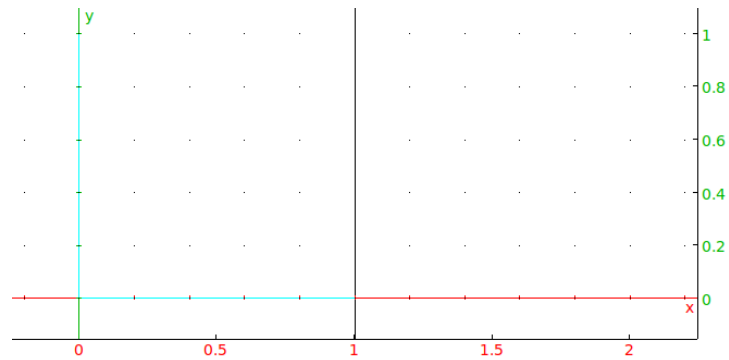
Output:



– *Input:*

```
line(x=1)
```

Output:



For a line in \mathbb{R}^3 :

- `line` takes two arguments:
`eqn1` and `eqn2`, two linear equations in the variables `x`, `y` and `z`.
- `line(eqn1,eqn2)` draws and returns the line which is the intersection of the planes given by the equations.

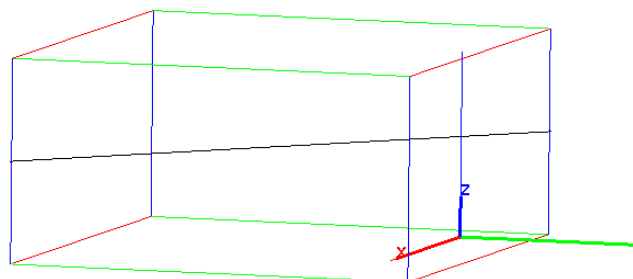
Examples.

– *Input:*

```
line(x+2*y+z-1=0,z=2)
```

Output:

```
mouse plan 0.814x+0.573y+0.094z=-0.0985
```

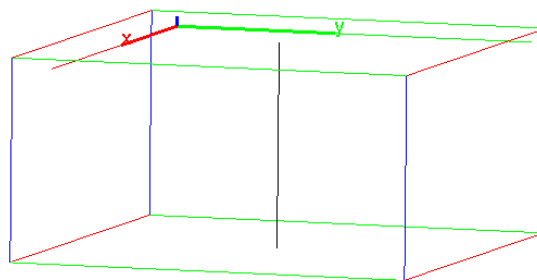


– *Input:*

```
line(y=1,x=1)
```

Output:

```
mouse plan 0.822x+0.289y+0.49z=0.866
```



Remark

`line` defines an oriented line:

- When a 2D line is given by an equation, it is rewritten as $lhs-rhs = ax + by + c = 0$, this determines its normal vector $[a, b]$ and the orientation is given by the vector $[b, -a]$.
- When a 3D line is given by two plane equations, its direction is defined by the cross product of the normals to the planes.

When the plane equation is rewritten as $lhs-rhs = ax + by + cz + d = 0$ the normal is $[a, b, c]$. For example, `line(x=y, y=z)` draws the line $x - y = 0, y - z = 0$ and its direction is:

$$[1, -1, 0] \times [0, 1, -1] = [1, 1, 1]$$

8.7.2 Drawing a 2D horizontal line: LineHorz

The `LineHorz` command draws a horizontal line in \mathbb{R}^2 .

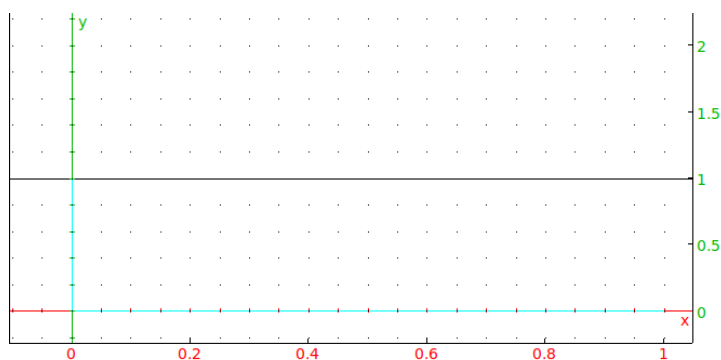
- `LineHorz` takes one argument:
 a , a number.
- `LineHorz(a)` draws the horizontal line $y = a$.

Example.

Input:

```
LineHorz(1)
```

Output:



8.7.3 Drawing a 2D vertical line: LineVert

The `LineVert` command draws a vertical line in \mathbb{R}^2 .

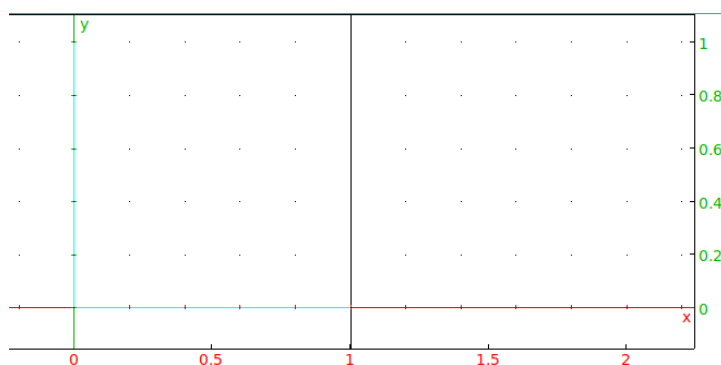
- `LineVert` takes one argument:
 a , a number.
- `LineVert(a)` draws the vertical line $x = a$.

Example.

Input:

`LineVert(1)`

Output:



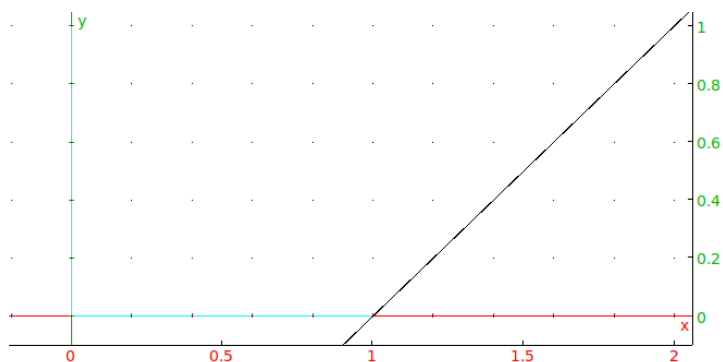
8.7.4 Tangent to a 2D graph: LineTan

The `LineTan` command draws tangent lines to graphs.

- `LineTan` takes two arguments:
 - * $expr$, in the variable x .
 - * x_0 , a value of x .
- `LineTan($expr, x_0$)` draws the tangent at $x = x_0$ to the graph of $expr$.

Example.*Input:*

```
LineTan(ln(x),1)
```

Output:*Input:*

```
equation(LineTan(ln(x),1))
```

Output:

$$y = (x - 1)$$

8.7.5 Tangent to a 2D graph: tangent

The `tangent` command draws tangents to surfaces.

- `tangent` takes two arguments:
 - * S , the graph of a two-variable function or a geometric object (see chapter 14).
 - * A , a point on S or a number (if S is a graph).
- `tangent(S, A)` draws tangent(s) to S passing through A .

Example.

Define the function `g`:

Input:

```
g(x):=x^2
```

then the graph `G` of `g` and a point `A` on the graph:

Input:

```
G:=plotfunc(g(x),x)::
A:=point(1.2,g(1.2))::
```


If you want to draw the tangent at the point **A** to the graph **G**, you can enter:

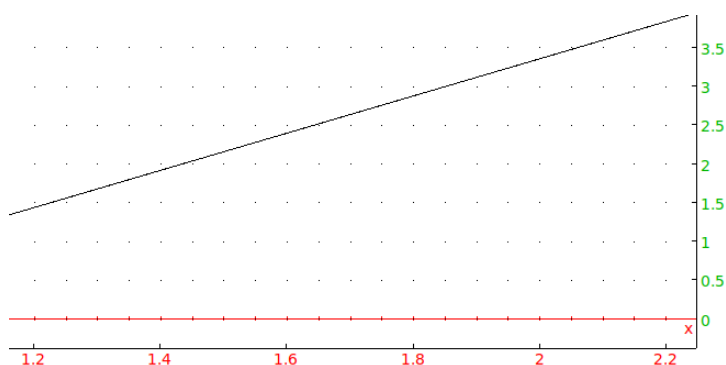
Input:

```
T:=tangent(G, A)
```

or:

```
T:=tangent(G, 1.2)
```

Output:



For the equation of the tangent line, you can enter:

Input:

```
equation(T)
```

Output:

$$y = 2.4x - 1.44$$

8.7.6 Plotting a line with a point and the slope: DrawSlp

The **DrawSlp** command can draw a line given a point and a slope.

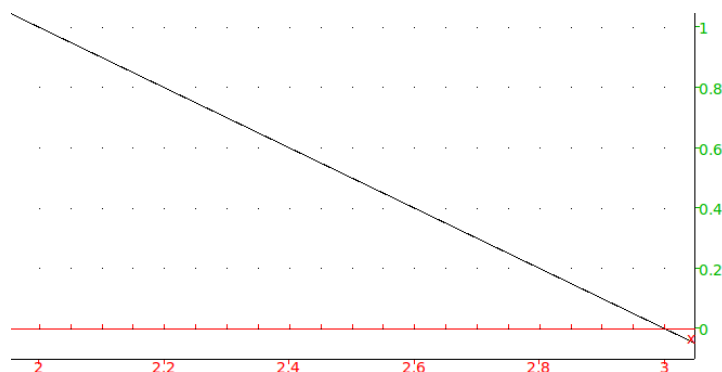
- **DrawSlp** takes three arguments:
 a , b and m , real numbers.
- **DrawSlp**(a, b, m) returns and draws the line through the point (a, b) with slope m .

Example.

Input:

```
DrawSlp(2,1,-1)
```

Output:



8.7.7 Intersection of a 2D graph with the axis

You can find the intersection of the graph $y = f(x)$ of a function with the axes using the commands covered so far.

- Finding the intersection of the graph with the y -axis is simply evaluating

$$f(0),$$

indeed the point with coordinates $(0, f(0))$ is the intersection point of the graph of f with the y -axis.

- Finding the intersection of the graph of f with the x -axis requires solving the equation $f(x) = 0$.

- * If $f(x)$ is polynomial-like, then you can find the exact values of the abscissa of these points with `solve` (see Section 6.55.6 p.619).

Input:

```
solve(f(x), x)
```

returns the solution.

- * Otherwise, you can find numeric approximations of these abscissa. First, look at the graph for an initial guess x_0 or a range with an intersection and then refine it with `fsolve` (see Section 10.4 p.818).

Input:

```
fsolve(f(x), x, x0, method)
```

returns a numeric approximation of a solution.

8.8 Graphing inequalities with two variables: `plotinequation` `inequationplot`

The `plotinequation` command plots the region of the plane where given inequalities hold.

- `plotinequation` takes two arguments:

- * *ineqs*, a list of inequalities in two variables.

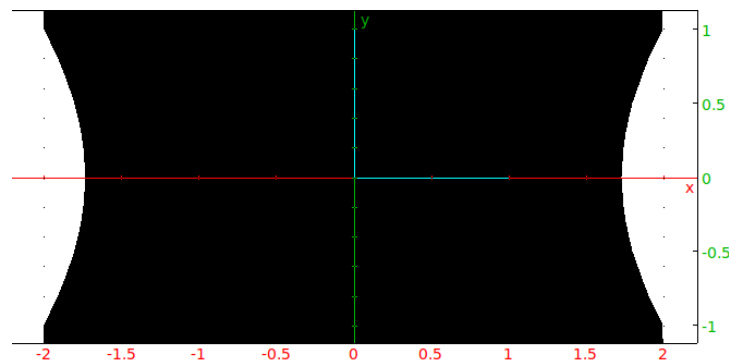
- * *vars*, a list of variables *var* or variables *var=a..b* with their ranges of values. Note that if the ranges are not specified, Xcas takes the default values of X-,X+,Y-,Y+ defined in the general graphic configuration (Cf **Graphic configuration**, see Section 3.5.8 p.76).
- *plotinequation(ineqs,vars)* draws the points of the plane whose coordinates satisfy the inequalities *ineqs*.

Examples.

– *Input:*

```
plotinequation(x^2-y^2<3,
[x=-2..2,y=-2..2],xstep=0.1,ystep=0.1)
```

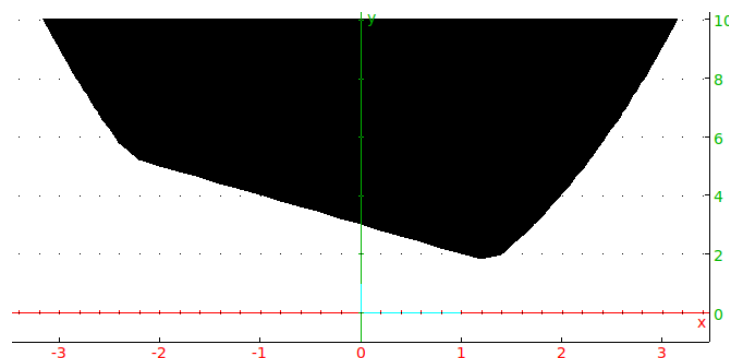
Output:



– *Input:*

```
plotinequation([x+y>3,x^2<y],
[x=-2..2,y=-1..10],xstep=0.2,ystep=0.2)
```

Output:



8.9 The area under a curve: area

The **area** command approximates the area under a graph.

- **area** takes four arguments:

- * *expr*, an expression $f(x)$.
- * *var=a..b*, the variable with a range.
- * *n*, a positive integer.
- * *method*, the approximation method to use, which can be one of:

- `trapezoid`
- `left_rectangle`
- `right_rectangle`
- `middle_point`
- `simpson`
- `rombergt` (Romberg with the trapezoid method)
- `rombergm` (Romberg with the midpoint method)
- `gauss15` (The 15 point Gaussian quadrature)

- `area(expr,var=a..b,n,method)` returns an approximation to the area under the graph over the given interval, using the specified method with n subdivisions (or 2^n subdivisions for `rombert`, `rombergm` and `gauss15`).

Examples.

- *Input:*

```
area(x^2,x=0..1,8,trapezoid)
```

Output:

```
0.3359375
```

- *Input:*

```
area(x^2,x=0..1,8,rombergm)
```

Output:

```
0.333333333333
```

- *Input:*

```
area(x^2,x=0..1,3,gauss15)
```

Output:

```
0.333333333333
```

- *Input:*

```
area(x^2,x=0..1)
```

Output:

$$\frac{1}{3}$$

8.10 Graphing the area below a curve: plotarea areaplot

The `plotarea` command draws the area below a graph. `areaplot` is a synonym for `plotarea`.

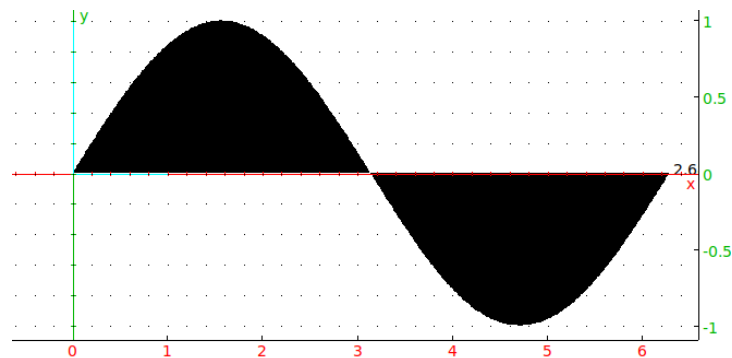
- `plotarea` takes two mandatory arguments and two optional arguments:
 - * `expr`, an expression representing the function to graph.
 - * `var=a..b`, the variable and the range of values.
 - * Optionally, `n`.
 - * Optionally, `method`, a method to approximate the region under the graph, which can be one of:
 - `trapezoid`
 - `rectangle_left`
 - `rectangle_right`
 - `middle_point`
- `plotarea(expr,var=a..b)` draws and shades the area between the graph of `expr` and the y -axis for $a < var < b$.
- `plotarea(expr,var=a..b[,n,method])` draws and shades the region used by the numeric approximation method `method` for area between the graph of `expr` and the y -axis for $a < var < b$, when $[a,b]$ is cut into n equal parts, along with the graph in red.

Examples.

- *Input:*

```
plotarea(sin(x),x=0..2*pi)
```

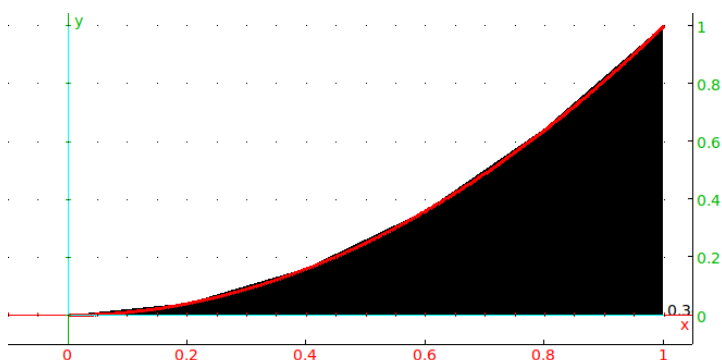
Output:



- *Input:*

```
plotarea(x^2,x=0..1,5,trapezoid)
```

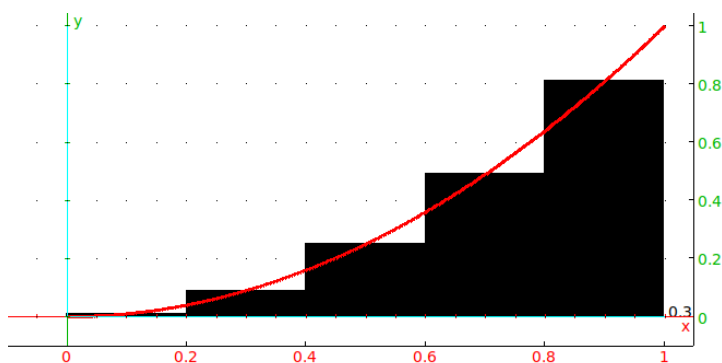
Output:



Input:

```
plotarea(x^2,x=0..1,5,middle_point)
```

Output:



8.11 Contour lines: plotcontour contourplot DrwCtour

The `plotcontour` command draws contour lines for functions of two variables.

`DrwCtour` and `contourplot` are synonyms for `plotcontour`.

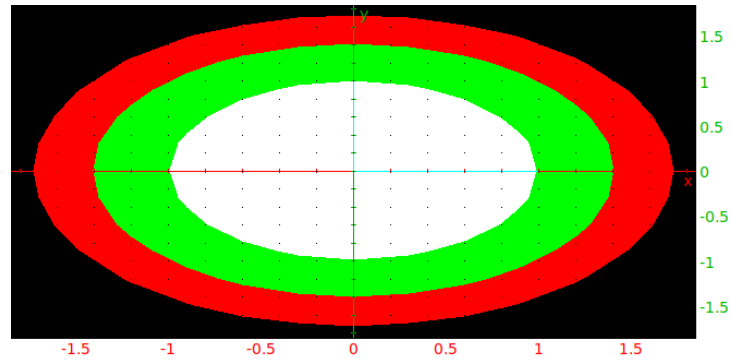
- `plotcontour` takes two mandatory arguments and one optional argument:
 - * *expr*, an expression involving two variables.
 - * *vars*, a list of the two variables.
 - * Optionally, *values*, a list of values of the contour lines to draw; *expr=value*, for *value* in *values* (by default, $[-10, -8, \dots, 8, 10]$).
- `plotcontour(expr,vars⟨, values⟩)` draws the contour lines *expr=value* for *value* in *values*.

Examples.

– *Input:*

```
plotcontour(x^2+y^2,[x=-3..3,y=-3..3],[1,2,3],
  display=[green,red,black]+[filled$3])
```

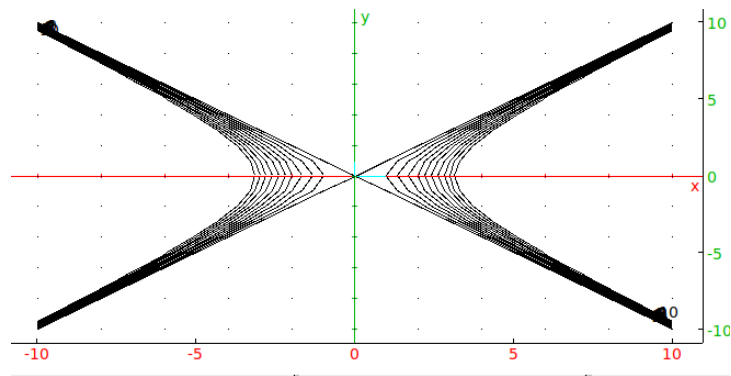
Output:



– *Input:*

```
plotcontour(x^2-y^2,[x,y])
```

Output:



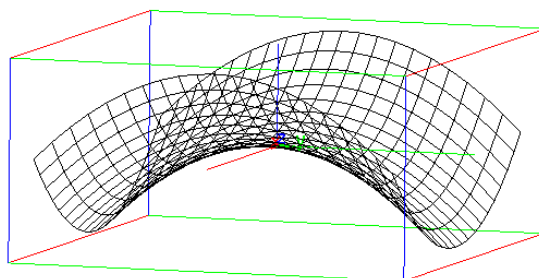
If you want to draw the surface in 3-d representation, you can use `plotfunc` (see Section 8.4.2 p.668).

Input:

```
plotfunc(x^2-y^2,[x,y])
```

Output:

mouse plan 0.943x+0.332y+0.0248z=0



8.12 2-d graph of a 2-d function with colors: plotdensity densityplot

The `plotdensity` command draws the graph of a function of two variables in the plane where the values of z are represented by the rainbow colors.

`densityplot` is a synonym for `plotdensity`.

- `plotdensity` takes two mandatory arguments and three optional arguments:
 - * *expr*, an expression of two variables.
 - * *vars*, a list of the variables and their ranges.
 - * Optionally, *z=a..b*, the range of z to correspond to the full rainbow (by default, it is deduced from the minimum and maximum value of *expr* on the discretization).
 - * Optionally, *xstep*, which can be `xstep=n` to specify the discretization step in the x direction.
 - * Optionally, *ystep*, which can be `ystep=m` to specify the discretization step in the y direction.
 - * Instead of *xstep* and *ystep*, you could use the option `nstep=n` to specify the number of points used to graph.
- `plotdensity(expr,vars⟨z=a..b,xstep,ystep⟩)` draws the graph of *expr* in the plane where the values of z are represented by the rainbow colors.

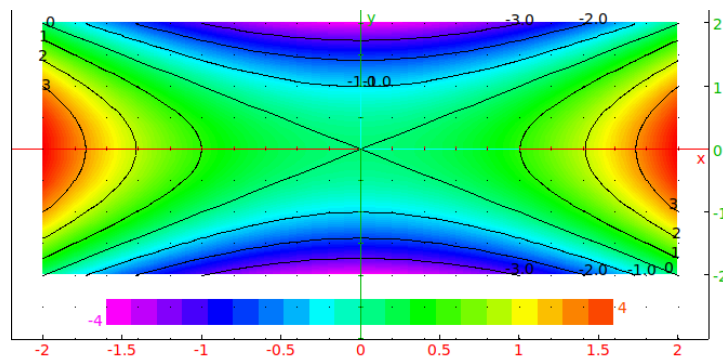
Remark: A rectangle representing the scale of colors will be displayed below the graph.

Example.

Input:

```
plotdensity(x^2-y^2,[x=-2..2,y=-2..2],
            xstep=0.1,ystep=0.1)
```

Output:



8.13 Implicit graph: plotimplicit implicitplot

The `plotimplicit` command draws curves or surfaces defined by an implicit expression or equation. If the option `unfactored` is given as the last argument, the original expression is taken unmodified. Otherwise, the expression is normalized, then replaced by the factorization of the numerator of its normalization.

Each factor of the expression corresponds to a component of the implicit curve or surface. For each factor, `Xcas` tests if it is of total degree less or equal to 2, in which case `conic` or `quadric` is called. Otherwise the numeric implicit solver is called.

Optional step and ranges arguments may be passed to the numeric implicit solver, note that they are dismissed for each component that is a conic or a quadric.

`implicitplot` is a synonym for `plotimplicit`.

8.13.1 2D implicit curve

For an implicit plot in \mathbb{R}^2 :

- `plotimplicit` takes three mandatory arguments and three optional arguments:
 - * *expr*, an expression of two variables implicitly defining a curve by $expr=0$.
 - * *vars*, a list of the two variables, optionally with their ranges $var=a..b$. If a range is not given, the ranges are determined by `WX-`, `WX+` and `WY-`, `WY+` in the graphical configuration (see Section 3.5.8 p.76).
 - * Optionally, *xstep*, which can be `xstep=n` to specify the discretization step in the x direction.
 - * Optionally, *ystep*, which can be `ystep=m` to specify the discretization step in the y direction.
 - * Optionally, `unfactored`.
- `plotimplicity(expr,vars(xstep,ystep,unfactored))` draws the graphic representation of the curve defined by the implicit equation $expr=0$ over the given ranges of the variables.

Examples.

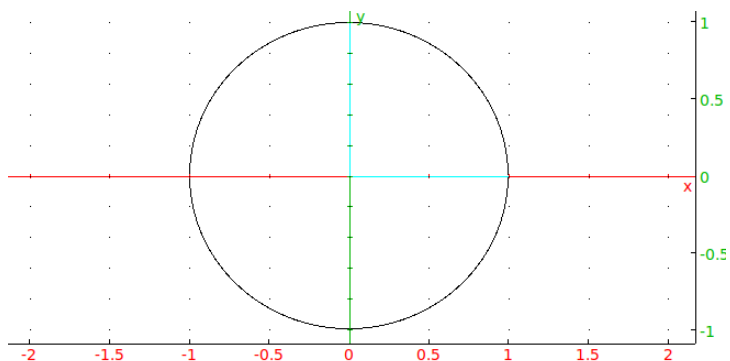
- *Input:*

```
plotimplicit(x^2+y^2-1,x,y)
```

or:

```
plotimplicit(x^2+y^2-1,x,y,unfactored)
```

Output:



– *Input:*

```
plotimplicit(x^2+y^2-1,x,y,xstep=0.2,ystep=0.3)
```

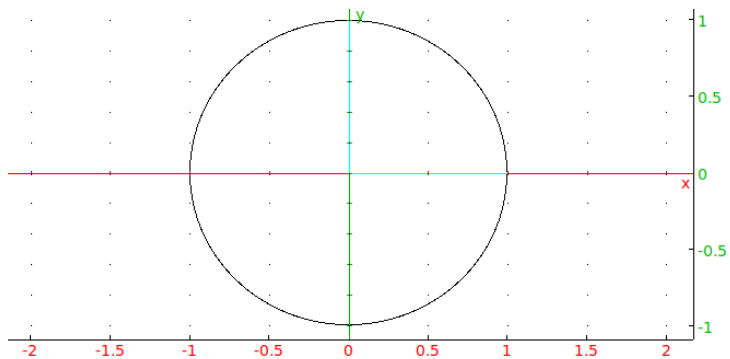
or:

```
plotimplicit(x^2+y^2-1,[x,y],xstep=0.2,ystep=0.3)
```

or:

```
plotimplicit(x^2+y^2-1,[x,y],
xstep=0.2,ystep=0.3,unfactored)
```

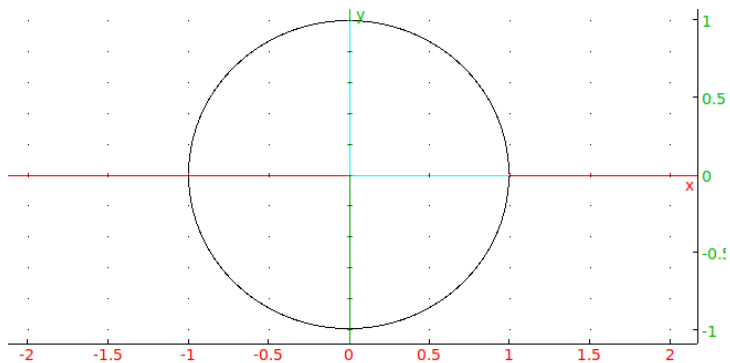
Output:



– *Input:*

```
plotimplicit(x^2+y^2-1,x=-2..2,y=-2..2,
xstep=0.2,ystep=0.3)
```

Output:



8.13.2 3D implicit surface

For an implicit plot in \mathbb{R}^2 :

- `plotimplicit` takes four mandatory arguments and three optional arguments:
 - * `expr`, an expression of three variables implicitly defining a curve by $expr=0$.
 - * `xvar`, `yvar` and `zvar`, the first, second and third variables, optionally with their ranges $var=a..b$.
 - * Optionally, `xstep`, which can be `xstep=n` to specify the discretization step in the x direction.
 - * Optionally, `ystep`, which can be `ystep=m` to specify the discretization step in the y direction.
 - * Optionally, `zstep`, which can be `zstep=p` to specify the discretization step in the z direction.
 - * Optionally, `unfactored`.
- `plotimplicit(expr,xvar,yvar,zvar [xstep,ystep,zstepunfactored])` draws the graphic representation of the surface defined by the implicit equation $expr=0$ over the given ranges of the variables.

Examples.

– *Input:*

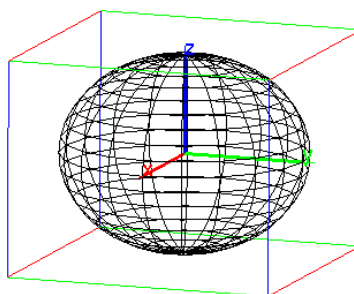
```
plotimplicit(x^2+y^2+z^2-1,x,y,z,
             xstep=0.2,ystep=0.1,zstep=0.3)
```

or:

```
plotimplicit(x^2+y^2+z^2-1,x,y,z,
             xstep=0.2,ystep=0.1,zstep=0.3,unfactored)
```

Output:

mouse pan 0.916x+0.322y+0.24z=0

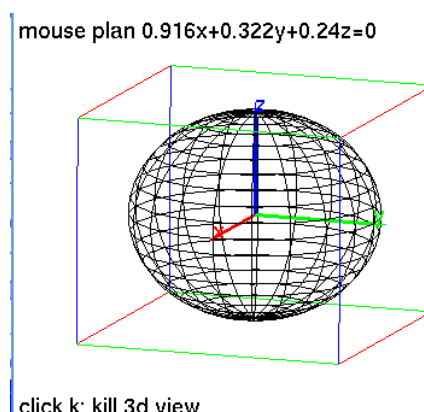


click k: kill 3d view

– *Input:*

```
plotimplicit(x^2+y^2+z^2-1,x=-1..1,y=-1..1,z=-1..1)
```

Output:



8.14 Parametric curves and surfaces: `plotparam` `paramplot` `DrawParm`

The `plotparam` command draws parametric curves and surfaces. `paramplot` and `DrawParm` are synonyms for `plotparam`.

8.14.1 2D parametric curve

To draw a parametric curve in \mathbb{R}^2 :

- `plotparam` takes two mandatory and one optional argument:
 - * *exprs*, a list of two real expressions or one complex expression involving the parameter.
 - * *var*, the parameter, optionally with a range *var=a..b*. If no range is given, the values of `t-` and `t+` from the graphics configuration will be used (see Section 3.5.8 p.76).
 - * Optionally, `tstep=n`, to set the discretization step.
- `plotparam(exprs,var[,tstep=n])` draws the parametric representation of the curve.

Examples.

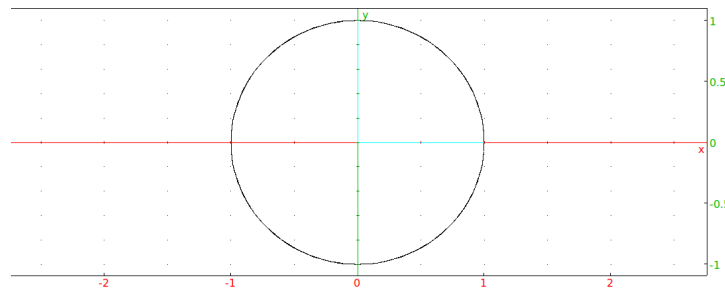
– *Input:*

```
plotparam(cos(x)+i*sin(x),x)
```

or:

```
plotparam([cos(x),sin(x)],x)
```

Output:



– *Input:*

```
plotparam(sin(t)+i*cos(t),t=-4..1)
```

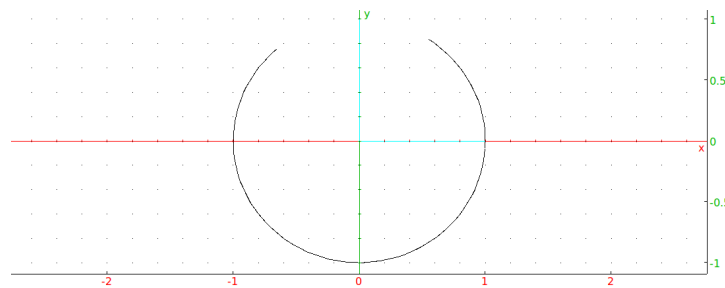
or:

```
plotparam(sin(x)+i*cos(x),x=-4..1)
```

or (with $t=-4, t+=1$ in the graphic configuration):

```
plotparam(sin(t)+i*cos(t))
```

Output:



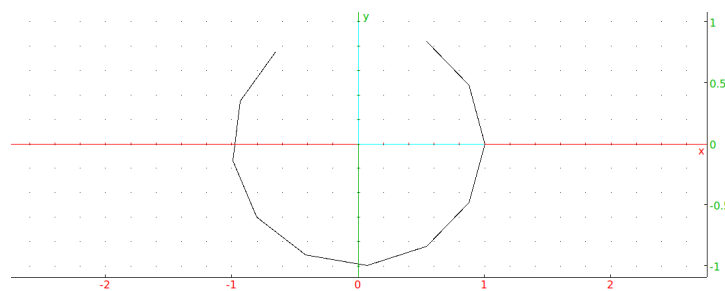
Input:

```
plotparam(sin(t)+i*cos(t),t=-4..1,tstep=0.5)
```

or (with $t=-4, t+=1$ in the graphic configuration):

```
plotparam(sin(t)+i*cos(t),t,tstep=0.5)
```

Output:



8.14.2 3D parametric surface: plotparam paramplot DrawPDM

To draw a parametric surface in \mathbb{R}^3 :

- `plotparam` takes two mandatory arguments and two optional arguments:
 - * *exprs*, a list of three expressions involving two parameters.
 - * *vars*, a list of the parameters, optionally with a range *var=a..b*.
 - * Optionally, `ustep=n`, to set the discretization step of the first parameter.
 - * Optionally, `vstep=m`, to set the discretization step of the second parameter.
- `plotparam(exprs,vars [, ustep=n, vstep=m])` draws the parametric representation of the surface.

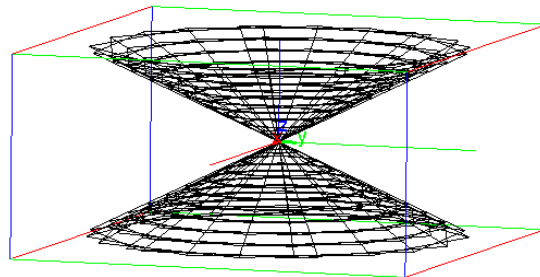
Examples.

– *Input:*

```
plotparam([v*cos(u),v*sin(u),v],[u,v])
```

Output:

mouse plan 0.915x+0.324y+0.24z=0

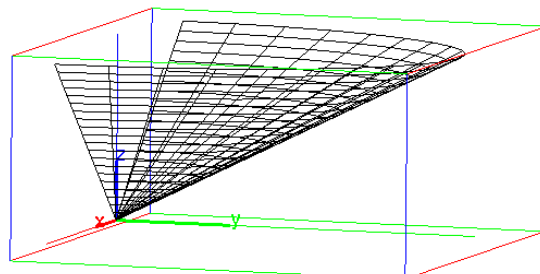


– *Input:*

```
plotparam([v*cos(u),v*sin(u),v],[u=0..pi,v=0..3])
```

Output:

mouse plan 0.767x+0.514y+0.384z=1.27

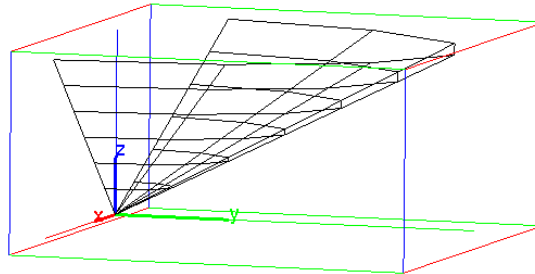


– *Input:*

```
plotparam([v*cos(u),v*sin(u),v],[u=0..pi,v=0..3],ustep=0.5,vstep=0.5)
```

Output:

mouse plan 0.768x+0.514y+0.382z=1.28



8.15 Bezier curves: bezier

The Bezier curve with the control points P_0, P_1, \dots, P_n is the curve parameterized by $\sum_{j=0}^n \binom{n}{j} t^j (1-t)^{n-j} P_j$. **bezier** plots Bezier curves.

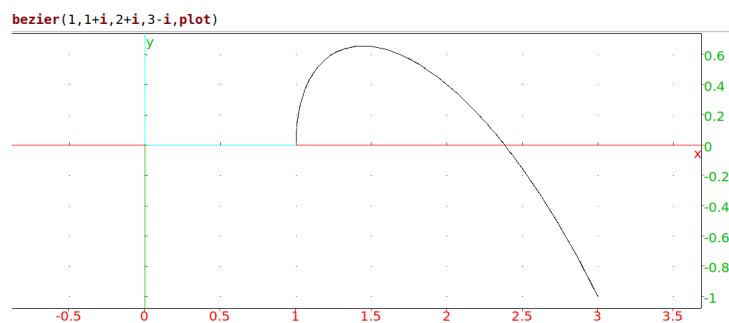
- **bezier** takes an unspecified number of arguments:
 - * *controls*, a sequence of control points.
 - * *plot*, the symbol.
- **bezier(controls,plot)** plots the Bezier curve with the given control points.

Examples.

- *Input:*

```
bezier(1,1+i,2+i,3-i,plot)
```

Output:



- *Input:*

```
bezier(point(0,0,0),point(1,1,0),point(0,1,1),plot)
```

Output:



To get the parameterization of the curve, you can use the `parameq` command (see Section 13.13.8 p.939).

Examples.

– *Input:*

```
parameq(bezier(1,1+i,2+i,3-i))
```

Output:

$$(1-t)^3 + 3t(1-t)^2(1+i) + 3t^2(1-t)(2+i) + t^3(3-i)$$

– *Input:*

```
parameq(bezier(point([0,0,0]),point([1,1,0]),point([0,1,1])))
```

Output:

$$[2t(1-t), 2t(1-t) + t^2, t^2]$$

8.16 Defining curves in polar coordinates: `plotpolar` `polarplot` `DrawPol` `courbe_polaire`

The `plotpolar` command draws a curve given in polar coordinates. `polarplot`, `DrawPol` and `courbe_polaire` are synonyms for `plotpolar`.

– `polarplot` takes two arguments:

- * *expr*, an expression involving a variable (which will represent the angle).
- * *var*, the variable. This can optionally include the range *var=a..b*.
- * Optionally, *tstep=n* to specify the discretization.

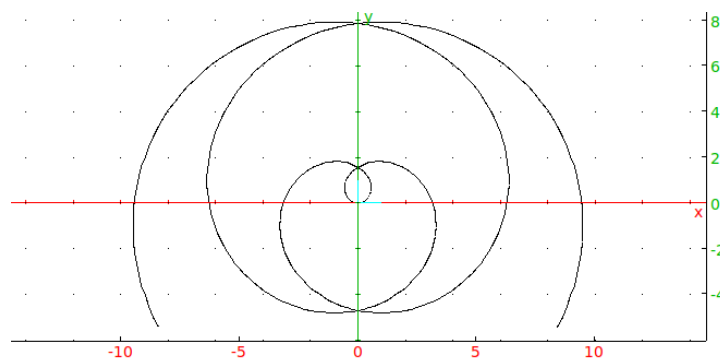
`polarplot(expr,var[,tstep=n])` draws the curve defined by $\rho = \text{expr}$ for $\theta = \text{var}$; in Cartesian coordinates that is the curve $(\text{expr} \cos \text{var}, \text{expr} \sin(\text{var}))$.

Examples.

– *Input*

`plotpolar(t,t)`

Output:



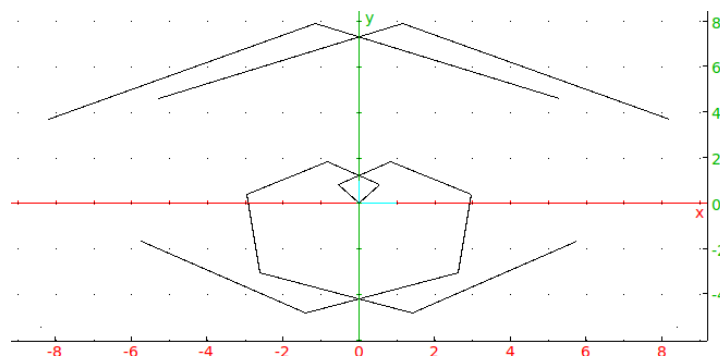
– *Input:*

`plotpolar(t,t,tstep=1)`

or:

`plotpolar(t,t=0..10,tstep=1)`

Output:



8.17 Graphing recurrent sequences: `plotseq` `seqplot` `graphe_suite`

The `plotseq` command draws the process of finding the terms of a recurrent sequence.

`seqplot` and `graphe_suite` are synonyms for `plotseq`.

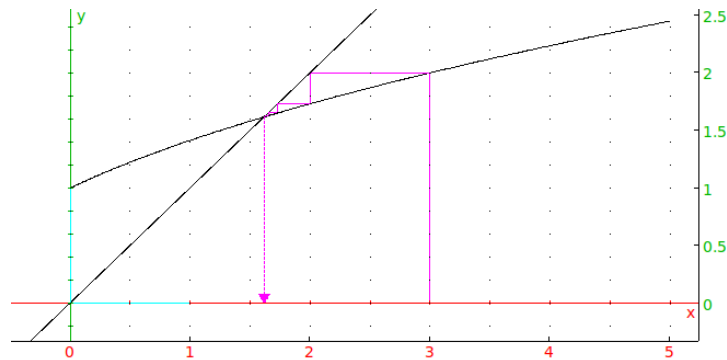
– `plotseq` takes :

- * *expr*, an expression depending on a variable.
- * *var=a*, the variable and a beginning value. (If *var=x*, then the variable name can be omitted.) The *a* value can be replaced by a list of three elements, $[a, x_-, x_+]$ where $x_- \dots x_+$ will be passed as the range of the variable for the graph computation.

- * n , the ending value of the variable.
- `plotseq(expr,var=a,n)` draws the line $y = x$, the graph of $y = \text{expr}$, and the n first terms of the recurrent sequence defined by: $u_0 = a$, $u_n = f(u_{n-1})$ where f is the function determined by expr .

Example.*Input:*

```
plotseq(sqrt(1+x),x=[3,0,5],5)
```

Output:**8.18 Tangent field: plotfield fieldplot**

The `plotfield` command draws the tangent field of a differential equation or a vector field.

`fieldplot` is a synonym for `plotfield`.

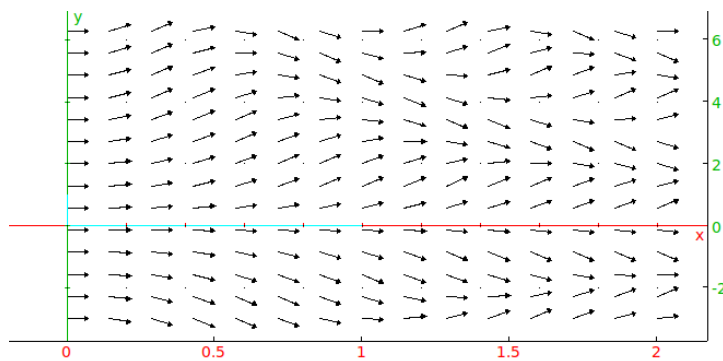
To draw the tangent field of a differential equation:

- `plotfield` takes two mandatory arguments and one optional argument:
 - * expr , an expression depending on two variables, a time variable and a dependent variable.
 - * vars , a list of the two variables $[t, y]$, where t is the time variable and y is the dependent variable. The variables can optionally include their ranges; $[t = a..b, y = c..d]$.
 - * Optionally, `ystep=n` to specify the discretization.
- `plotfield(expr,vars<,ystep=n)` draws the tangent field of the differential equation $y' = f(t, y)$.

Example.*Input:*

```
plotfield(4*sin(t*y),[t=0..2,y=-3..7])
```

Output:



To draw a vector field:

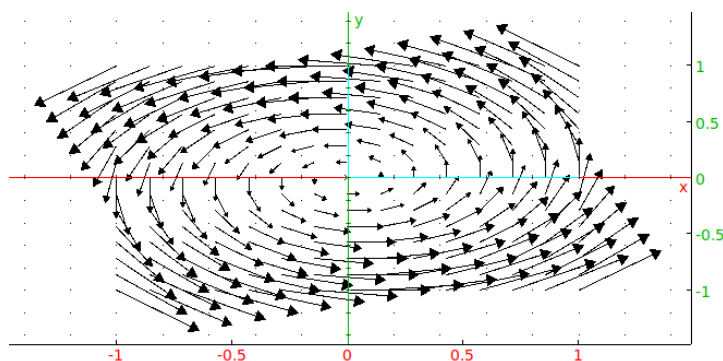
- `plotfield` takes two mandatory arguments and two optional arguments:
 - * V , a list of two expressions involving two variables.
 - * $vars$, a list of the two variables. The variables can optionally include their ranges $var=a..b$.
 - * Optionally, `xstep=n` to specify the discretization of the first variable.
 - * Optionally, `ystep=m` to specify the discretization of the second variable.
- `plotfield($V, vars$, [xstep=n, ystep=m])` draws the vector field given by V .

Example.

Input:

```
plotfield(5*[-y,x], [x=-1..1,y=-1..1])
```

Output:



8.19 Plotting a solution of a differential equation: `plotode odeplot`

The `plotode` command draws solutions of differential equations.

- **plotode** takes three mandatory arguments and one optional argument:
 - * *expr* an expression depending on two or three variables, a time variable and one or two dependent variables.
 - * *vars*, a list of the time variable and the dependent variable. The time variable can optionally have a range of values, such as $t = a..b$. The dependent variable can also be a vector of size two.
 - * *init*, the initial values of the variables.
 - * Optionally, for when there are two dependent variables, **plane**, the symbol, to draw the solution in the plane.
- **plotode**(*expr*,*vars*,*init*) draws the solution of the differential equation $y' = \text{expr}$ (where y is the dependent variable) passing through the initial point *init*.

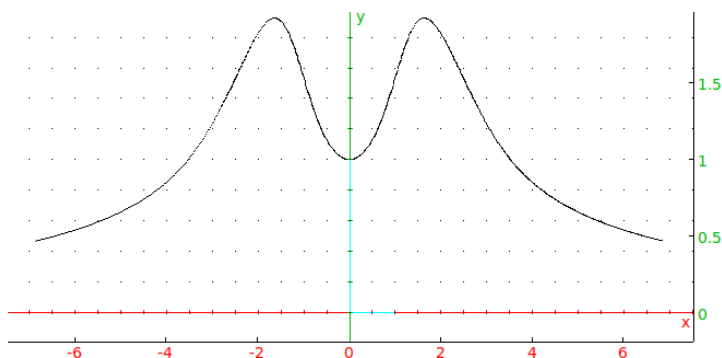
To compute the values of the solutions, see Section 10.3.5 p.812.

Examples.

– *Input:*

```
plotode(sin(t*y),[t,y],[0,1])
```

Output:

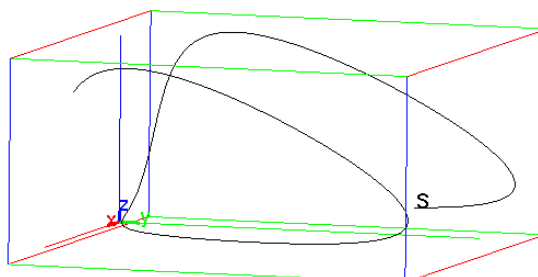


– *Input:*

```
S:=plotode([h-0.3*h*p, 0.3*h*p-p],
[t,h,p],[0,0.3,0.7])
```

Output:

mouse plan 0.965x+0.211y+0.157z=3.14

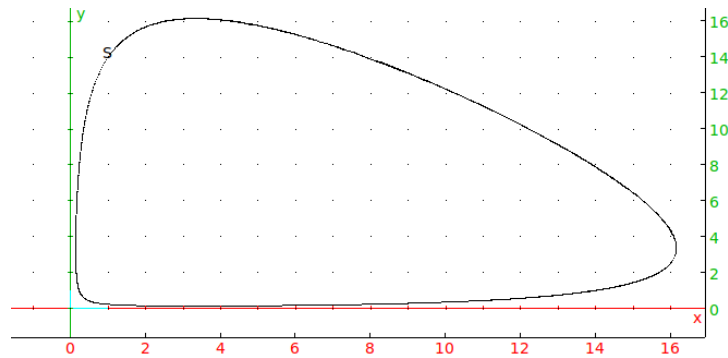


8.20. INTERACTIVE PLOTTING OF SOLUTIONS OF A DIFFERENTIAL EQUATION: INTERACTIVE_

– *Input (for a 2-d graph in the plane):*

```
S:=odeplot([h-0.3*h*p, 0.3*h*p-p],
           [t,h,p],[0,0.3,0.7],plane)
```

Output:



8.20 Interactive plotting of solutions of a differential equation: interactive_plotode interactive_odeplot

The `interactive_plotode` command draws interactive tangent fields of differential equations.

`interactive_odeplot` is a synonym for `interactive_plotode`.

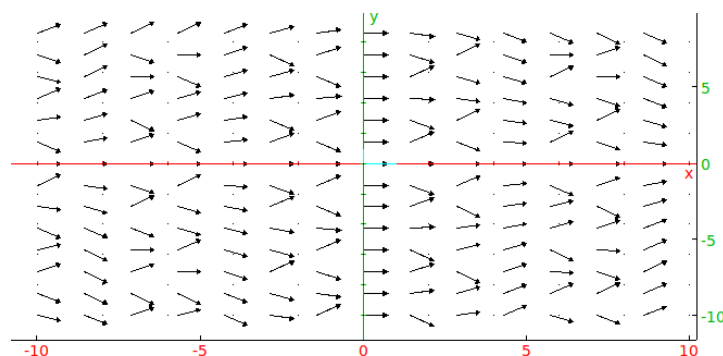
- `interactive_plotode` takes two arguments:
 - * *expr* an expression depending on two or three variables, a time variable and one or two dependent variables.
 - * *vars*, a list of the time variable and the dependent variable.
- `interactive_plotode(expr,vars)` draws the tangent field of the differential equation $y' = \text{expr}$ (where y is the dependent variable) in a new window. In this window, one can click on a point to get the plot of the solution of $y' = \text{expr}$ passing through this point. You can further click to display several solutions. To stop, press the Esc key.

Example.

Input:

```
interactive_plotode(sin(t*y),[t,y])
```

Output:



Solutions of the differential equation can be plotted by clicking on an initial point.

8.21 Animated graphs (2D, 3D or "4D")

Xcas can display animated 2D, 3D or "4D" graphs. This is done first by computing a sequence of graphic objects, then after completion, by displaying the sequence in a loop. To stop or start again the animation, click on the button ►| (to the left of Menu).

• 8.21.1 Animation of a 2D graph: `animate`

The `animate` command creates two-dimensional animations using graphs of functions depending on a parameter. (See also Section 8.5 p.673.)

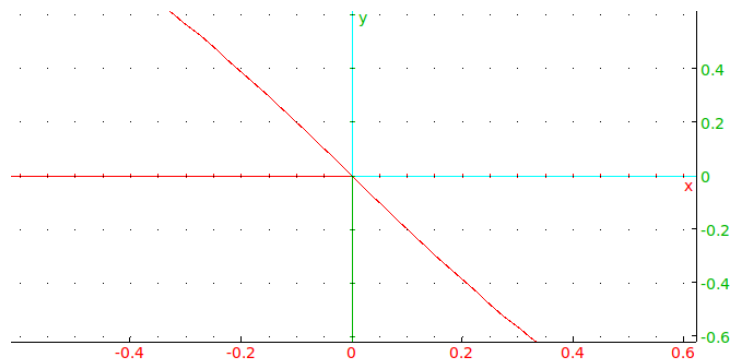
- `animate` takes three mandatory arguments and two optional arguments:
 - * `expr`, an expression involving two variables, one of which will be regarded as the parameter.
 - * `var` the name of the (non-parameter) variable in the expression, which can also specify a range of values `var=a..b`.
 - * `param` the name of the parameter, which can also specify a range of values.
 - * Optionally, `frames=n`, where n is an integer specifying the number of frames.
 - * Optionally, `opt`, which can be `xstep=n` to specify the discretization step or `nstep=n` to specify the number of points used to graph.
- `animate(expr,var,param,frames=n(opt))` draws an animation consisting of graph of the function as the parameter varies.

Examples.

```
animate(sin(a*x),x=-pi..pi,a=-2..2,frames=10,color=red)
```

Output:

The output is an animation beginning with:



8.21.2 Animation of a 3D graph: `animate3d`

The `animate3d` command creates three-dimensional animations using graphs of functions depending on a parameter. (See also Section 8.4.2 p.668.)

- * `animate` takes three mandatory arguments and two optional arguments:
 - *expr*, an expression involving three variables, one of which will be regarded as the parameter.
 - *vars* a list of the the (non-parameter) variables in the expression, which can also specify ranges of values *var=a..b*.
 - *param* the name of the parameter, which can also specify a range of values.
 - Optionally, `frames=n`, where *n* is an integer specifying the number of frames.
 - Optionally, *xstep*, which can be `xstep=n` to specify the discretization step in the *x* direction.
 - Optionally, *ystep*, which can be `ystep=m` to specify the discretization step in the *y* direction.
 - Instead of *xstep* and *ystep*, you could use the option `nstep=n` to specify the number of points used to graph.
- * `animate3d(expr,vars,param,frames=n<opt,xstep=n, ystep=m>)` draws an animation consisting of graph of the function as the parameter varies.

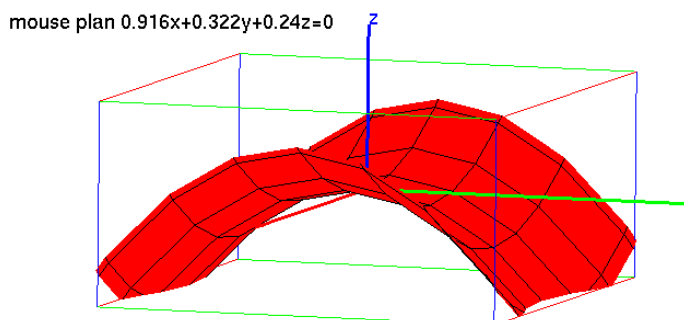
Example.

Input:

```
animate3d(x^2+a*y^2,[x=-2..2,y=-2..2],a=-2..2,
          frames=10,display=red+filled)
```

Output:

The output is an animation beginning with:



8.21.3 Animation of a sequence of graphic objects: animation

The `animation` command creates animations using sequences of graphic objects, which can be graphs (see Section 8.4 p.667) or not (see chapters 13 and 14). The sequence of objects depends most of the time on a parameter and is defined using the `seq` command but it is not mandatory. To define a sequence of graphic objects with `seq`, enter the definition of the graphic object (depending on the parameter), the parameter name, its minimum value, its maximum value maximum and optionally a step value.

- * `animation` takes:
 - *objs*, a sequence of graphic objects.
- * `animation(objs)` draws an animation consisting of the sequence of objects.

Examples.

* *Input:*

```
animation(seq(plotfunc(cos(a*x),x),a,0,10))
```

Output:

The output is an animation beginning with:



* *Input:*

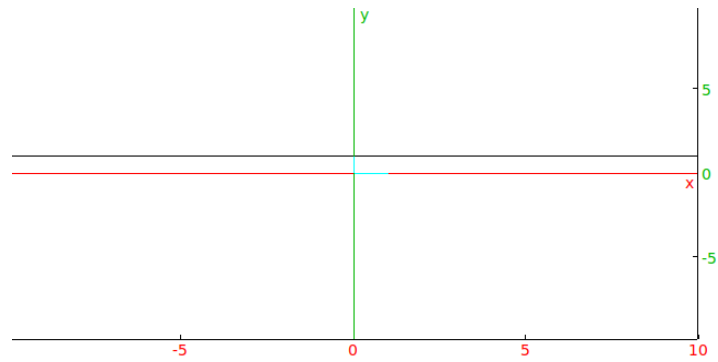
```
animation(seq(plotfunc(cos(a*x),x),a,0,10,0.5))
```

or:

```
animation(seq(plotfunc(cos(a*x),x),a=0..10,0.5))
```

Output:

The output is an animation beginning with:

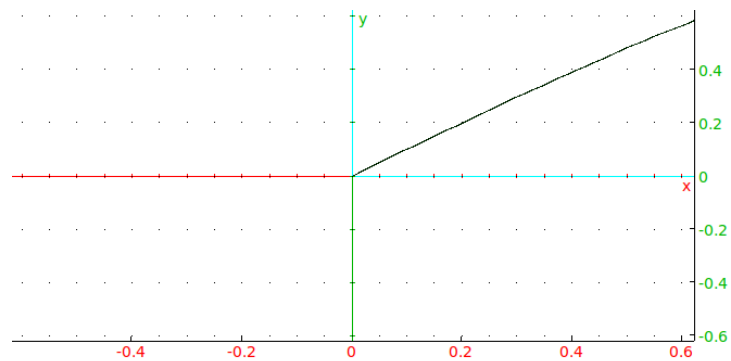


* *Input:*

```
animation(seq(plotfunc([cos(a*x),sin(a*x)],x=0..2*pi/a),
               a,1,10))
```

Output:

The output is an animation beginning with:

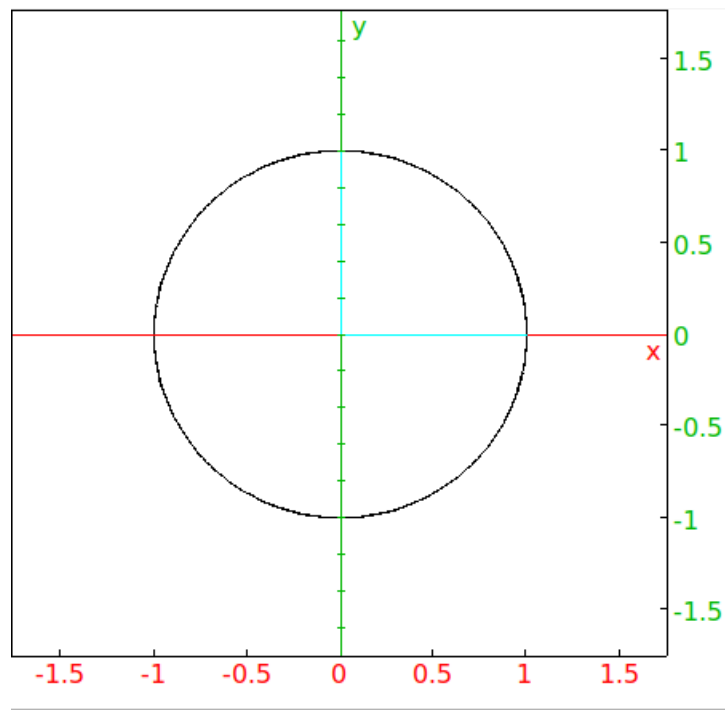


* *Input:*

```
animation(seq(plotparam([cos(a*t),sin(a*t)]),
               t=0..2*pi),a,1,10))
```

Output:

The output is an animation beginning with:

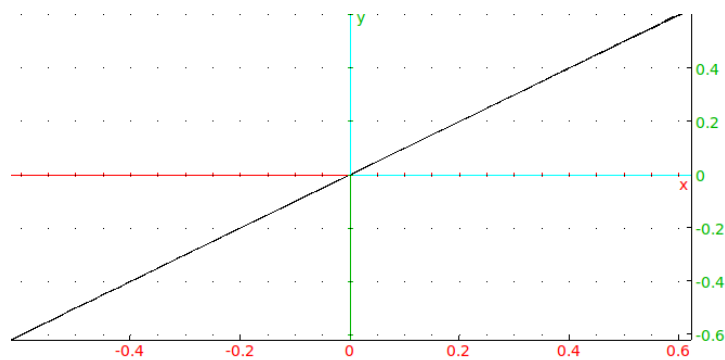


* *Input:*

```
animation(seq(plotparam([sin(t),sin(a*t)]),
  t,0,2*pi,tstep=0.01),a,1,10))
```

Output:

The output is an animation beginning with:

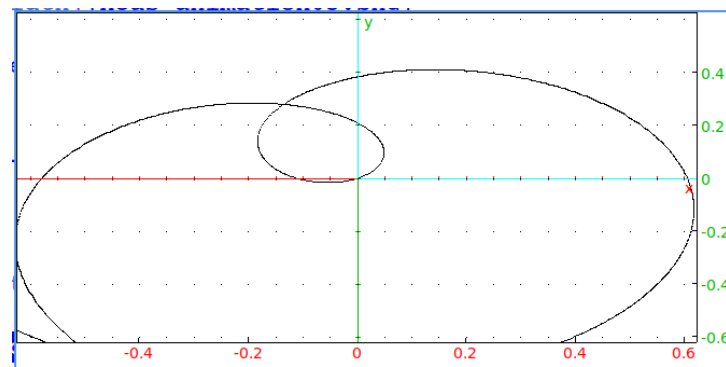


* *Input:*

```
animation(seq(plotpolar(1-a*0.01*t^2,
  t,0,5*pi,tstep=0.01),a,1,10))
```

Output:

The output is an animation beginning with:

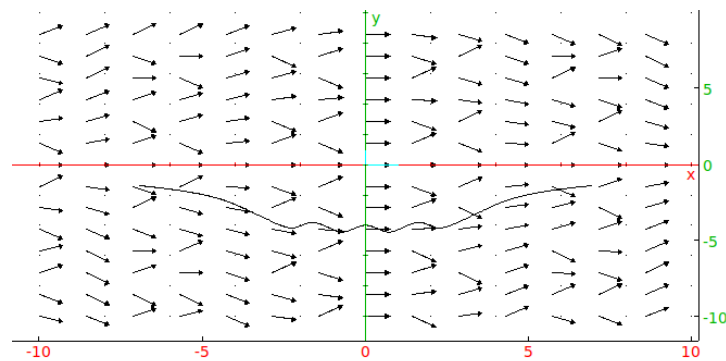


* *Input:*

```
plotfield(sin(x*y),[x,y]);
animation(seq(plotode(sin(x*y),[x,y],[0,a]),a,-4,4,0.5))
```

Output:

The output is an animation beginning with:

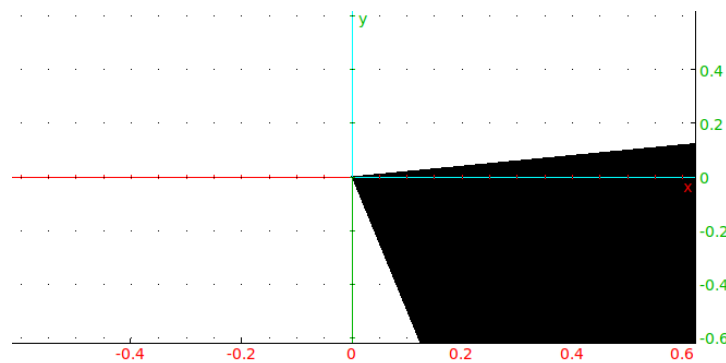


* *Input:*

```
animation(seq(display(square(0,1+i*a),filled),a,-5,5))
```

Output:

The output is an animation beginning with:



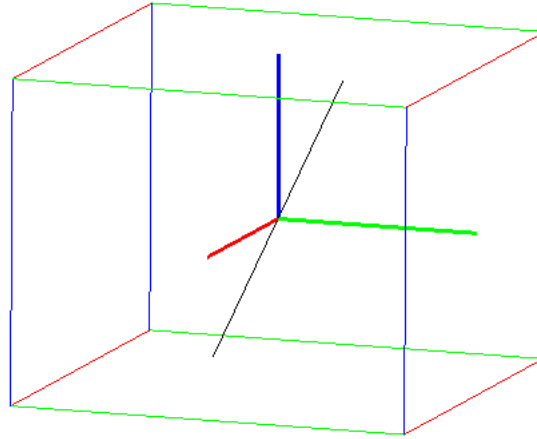
* *Input:*

```
animation(seq(line([0,0,0],[1,1,a]),a,-5,5))
```

Output:

The output is an animation beginning with:

mouse plan 0.916x+0.322y+0.24z=0



click k: kill 3d view

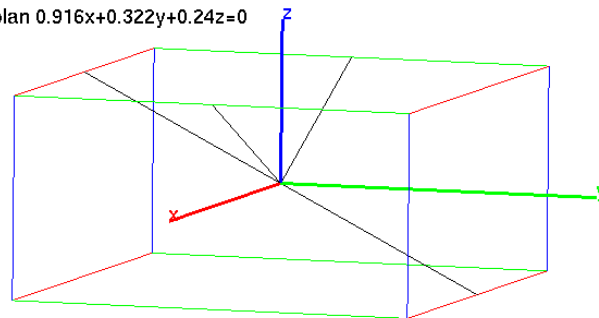
* *Input:*

```
animation(seq(plotfunc(x^2-y^a,[x,y]),a=1..3))
```

Output:

The output is an animation beginning with:

mouse plan 0.916x+0.322y+0.24z=0



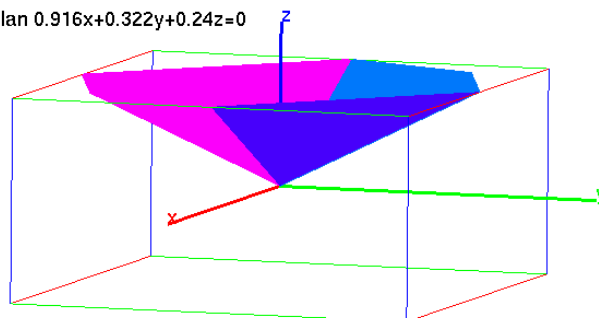
* *Input:*

```
animation(seq(plotfunc((x+i*y)^a,[x,y],
display=filled),a=1..10))
```

Output:

The output is an animation beginning with:

mouse plan 0.916x+0.322y+0.24z=0



Remark: You can also define the sequence with a program. For example if you want to draw the segments of length $1, \sqrt{2} \dots \sqrt{20}$

constructed with a right triangle of side 1 and the previous segment (note that there is a `c:=evalf(...)` statement to force approximate evaluation otherwise the computing time would be too long):

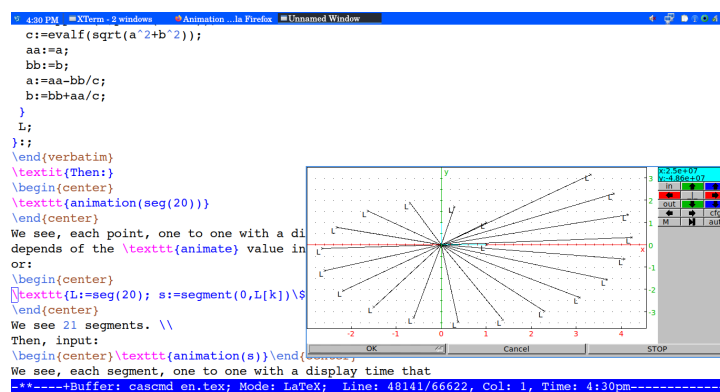
Input:

```
seg(n):={
  local a,b,c,j,aa,bb,L;
  a:=1;
  b:=1;
  L:=point(1);
  for(j:=1;j<=n;j++){
    L:=append(L,point(a+i*b));
    c:=evalf(sqrt(a^2+b^2));
    aa:=a;
    bb:=b;
    a:=aa-bb/c;
    b:=bb+aa/c;
  }
  L;
}::
```

then:

```
L:=seg(20); s:=segment(0,L[k])$(k=0..20)
```

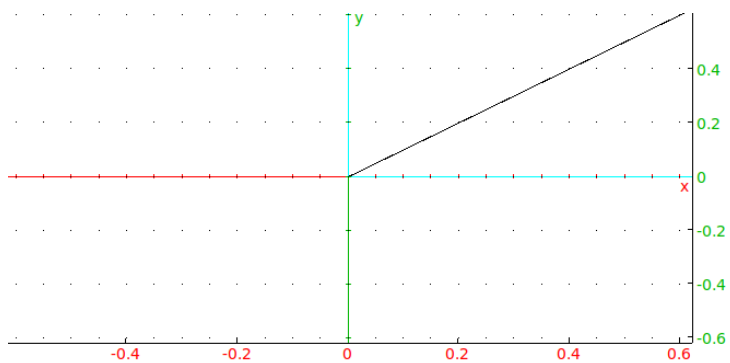
Output:



then:

`animation(s)`

The output is an animation displaying the segments one at a time, beginning with:



Chapter 9

Statistics

9.1 One variable statistics

Xcas has several functions to perform statistics; the data is typically given as a list of numbers, such as $A := [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]$. This particular list will be used in several examples. Section 6.45.16 p.535 will discuss statistics on matrices.

9.1.1 The mean: mean

Recall that the mean of a list x_1, \dots, x_n is simply their numeric average $(x_1 + \dots + x_n)/n$. The **mean** command finds the mean of a list.

- * **mean** takes one mandatory argument and one optional argument:
 - L , a list or matrix of numbers.
 - W , a list or matrix of weights, the same size as L .
- * **mean(L)W()** returns the mean of the list or a list with the means of the columns of the matrix.

Examples.

* *Input:*

mean([1, 2, 3, 4])

Output:

$\frac{5}{2}$

since $(1 + 2 + 3 + 4)/4 = 5/2$.

* *Input:*

mean([1, 2, 3], [5, 6, 7])

Output:

[3, 4, 5]

since $(1 + 5)/2 = 3$, $(2 + 6)/2 = 4$ and $(3 + 7)/2 = 5$.

* *Input:*

`mean([2,4,6,8],[2,2,3,3])`

Output:

$$\frac{27}{5}$$

since $(2 \cdot 2 + 4 \cdot 2 + 6 \cdot 3 + 8 \cdot 3)/(2 + 2 + 3 + 3) = 27/5$.

* *Input:*

`mean([[1,2],[3,4]],[[1,2],[2,1]])`

Output:

$$\begin{bmatrix} \frac{7}{3} & \frac{8}{3} \end{bmatrix}$$

since $(1 \cdot 1 + 3 \cdot 2)/(1 + 2) = 7/3$ and $(2 \cdot 2 + 4 \cdot 1)/(2 + 1) = 8/3$.

9.1.2 Variance: variance

The variance of a list of numbers measures how close the numbers are to their mean by finding the average of the squares of the differences between the numbers and the mean; specifically, given a list of numbers $[x_1, \dots, x_n]$ with mean $\mu = (x_1 + \dots + x_n)/n$, the variance is

$$\frac{(x_1 - \mu)^2 + \dots + (x_n - \mu)^2}{n}.$$

The squares help ensure that the numbers above the mean and those below the mean don't cancel out. The **variance** command computes the variance.

* **variance** takes one mandatory argument and one optional argument:

- L , a list or matrix of numbers.
- W , a list or matrix of weights, the same size as L .

* **variance**(L) W () returns the variance of the list or a list with the variances of the columns of the matrix.

Examples.

* *Input:*

`variance([1,2,3,4])`

Output:

$$\frac{5}{4}$$

* *Input:*

`variance([[1,2,3],[5,6,7]])`

Output:

$$[4, 4, 4]$$

* *Input:*

`variance([2,4,6,8],[2,2,3,3])`

Output:

$$\frac{121}{25}$$

* *Input:*

`variance([[1,2],[3,4]],[[1,2],[2,1]])`

Output:

$$\begin{bmatrix} 8 & 8 \\ \frac{8}{9} & \frac{8}{9} \end{bmatrix}$$

9.1.3 Standard deviation: `stdev`

Standard deviation is potentially better than variance to measure how close numbers are to their mean. The standard deviation is the square root of the variance; for example, the list $[1, 2, 3, 4]$ has mean $5/2$, and so the standard deviation will be $2\sqrt{5}/4$, since

$$\sqrt{\frac{(1 - 5/2)^2 + (2 - 5/2)^2 + (3 - 5/2)^2 + (4 - 5/2)^2}{4}} = \frac{2\sqrt{5}}{4}$$

Note that if the list of numbers have units, then the standard deviation will have the same unit.

The `stddev` command finds the standard deviation.

* `stddev` takes one mandatory argument and one optional argument:

- L , a list or matrix of numbers.
- W , a list or matrix of weights, the same size as L .

* `stddev(L)W()` returns the standard deviation of the list or a list with the standard deviations of the columns of the matrix.

Examples.

* *Input:*

`stddev([1,2,3,4])`

Output:

$$\frac{\sqrt{5}}{2}$$

* *Input:*

`stddev([1,2,3],[2,1,1])`

Output:

$$\frac{4}{16}\sqrt{11}$$

* *Input:*

`stddev([[1,2],[3,6]])`

Output:

$$[1, 2]$$

9.1.4 The population standard deviation: `stddev` `stdDev`

Given a large population, rather than collecting all of the numbers it might be more feasible to get a smaller collection of numbers and try to extrapolate from that. For example, to get information about the ages of a large population, you might get the ages of a sample of 100 of the people and work with that.

If a list of numbers is a sample of data from a larger population, then the mean of the sample can be used to estimate the mean of the population. The standard deviation uses the mean to find the standard deviation of the sample, but since the mean of the sample is only an approximation to the mean of the entire population, the standard deviation of the sample doesn't provide an optimal estimate of the standard deviation of the population. An unbiased estimate of the standard deviation of the entire population is given by the population standard deviation; given a list $L = [x_1, \dots, x_n]$ with mean μ , the population standard deviation is

$$s = \sqrt{\frac{(x_1 - \mu)^2 + \dots + (x_n - \mu)^2}{n - 1}}.$$

Note that

$$s^2 = \frac{n}{n - 1} \sigma^2.$$

where σ is the standard deviation of the sample.

The `stddev` command finds the standard deviation.

`stdDev` is a synonym for `stddev`, for TI compatibility. There is no population variance function; if needed, it can be computed by squaring the `stddev` function.

- * `stddev` takes one mandatory argument and one optional argument:
 - L , a list or matrix of numbers.
 - W , a list or matrix of weights, the same size as L .
- * `stddev(L)W()` returns the population standard deviation of the list or a list with the population standard deviations of the columns of the matrix.

Examples.

* *Input:*

`stddev([1,2,3,4])`

Output:

$$\frac{\sqrt{5}}{2}$$

while:

Input:

`stddevp([1,2,3,4])`

Output:

$$\frac{\sqrt{15}}{3}$$

* *Input:*

A:= [0,1,2,3,4,5,6,7,8,9,10,11] stddevp(A,A)

Output:

$$\frac{\sqrt{66}}{3}$$

9.1.5 The median: median

Although the average of a list of numbers typically means the mean, there are other notions of “average”. Another such notion is the median; the median of a list of numbers is the middle number when they are listed in numeric order. For example, the median of the list [1, 2, 5, 7, 20] is simply 5. If the length of a list of numbers is even, so there isn’t a middle number, the median is then the mean of the two middle numbers; for example, the median of [1, 2, 5, 7, 20, 21] is $(5 + 7)/2 = 6$.

The **median** function finds the median of a list.

* **median** takes one mandatory argument and one optional argument:

- L , a list of numbers.
- Optionally, W , a list of positive integers for weights, where the weight of number represents how many times it is counted in a list.

* **median**(L [W]) returns the median of the list.

Examples.

* *Input:*

median([1,2,5,7,20])

Output:

5

* *Input:*

median([1,2,5,7,20], [5,3,2,1,2])

Output:

2

since the median of 1, 1, 1, 1, 1, 2, 2, 2, 5, 5, 7, 20, 20 is 2.

9.1.6 Quartiles: quartiles quartile1 quartile3

Recall that the quartiles of a list of numbers divide it into four equal parts; the first quartile is the number q_1 such that one-fourth of the list numbers fall below q_1 ; i.e., the median of that part of the list which fall at or below the list median. The second quartiles is the number q_2 such that half of the list numbers fall

at or below q_2 ; more specifically, the median of the list. And of course the third quartile is the number q_3 such that three-fourths of the list numbers fall at or below q_3 .

The function **quartiles** finds the minimum of a list, the first quartile, the second quartile, the third quartile and the maximum of the list.

* **quartiles** takes one mandatory argument and one optional argument:

- L , a list of numbers.
- Optionally, W , a list of weights.

* **quartiles**(L (W)) returns a column vector consisting of the minimum, first second and third quartile, and the maximum of L .

The **min**, **quartile1**, **median**, **quartile3** and **max** commands find the individual entries of this list.

Example.

Input:

```
A:= [0,1,2,3,4,5,6,7,8,9,10,11]; quartiles(A)
```

Output:

$$\begin{bmatrix} 0.0 \\ 2.0 \\ 5.0 \\ 8.0 \\ 11.0 \end{bmatrix}$$

Input:

```
min(A),quartile1(A),median(A),quartile3(A),max(A)
```

Output:

```
0, 2.0, 5.0, 8.0, 11
```

– *Input:*

```
quartiles(A,A)
```

Output:

```
[0, 6, 8, 10, 11]
```

9.1.7 Quantiles: quantile

Similar to quartiles, a quantile of a list is the number q such that a given fraction of the list numbers fall at or below q . The first quartile, for example, is the quantile with the fraction 0.25.

The **quantile** command finds quantiles.

- **quantile** takes two mandatory arguments and one optional argument:

- * L , a list of numbers.
- * Optionally, W , a list of weights.
- * p , a number between 0 and 1.

`quantile(L, p)` returns the p th quantile of L .

Examples.

– *Input:*

```
A:= [0,1,2,3,4,5,6,7,8,9,10,11]
quantile(A,0.1)
```

Output:

1.0

– *Input:*

```
quantile(A,A,0.25)
```

Output:

6

9.1.8 The boxwhisker: boxwhisker mustache

A boxwhisker is a graphical view of the quartiles of a list of numbers. The boxwhisker consists of a line segment from the the minimum of the list to the first quartile, leading to a rectangle from the first quartile to the third quartile, followed by a line segment from the third quartile to the maximum of the list. The rectangle contains a vertical segment indicating the median, and the two line segments will contain vertical lines indicating the first and ninth decile.

The `boxwhisker` command creates a boxwhisker for a list. `mustache` is a synonym for `boxwhisker`.

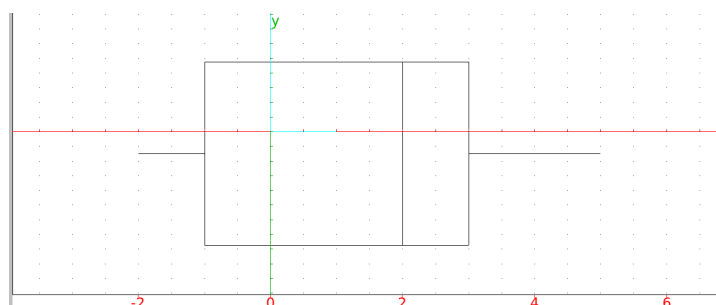
- `boxwhisker` takes one argument:
 L , a list of numbers.
- `boxwhisker(L)` draws the boxwhisker for L .

Example.

Input:

```
boxwhisker([-1,1,2,2.2,3,4,-2,5])
```

Output:



9.1.9 Classes: `classes`

The `classes` command groups a collection of numbers into intervals.

- `classes` takes two or three arguments:
 - * L , a list or matrix of numbers.
 - * Optionally, a and b , numbers. (By default, these will be `class_min` and `class_size` from the graphics configuration, see Section 3.5.8 p.76, which themselves default to 0 and 1.)
 - or:**
 - * Optionally, a and M , where a is the start of the beginning interval and M consists of the midpoints of the intervals.
 - or:**
 - * Optionally, I , a list of intervals to use.
- `classes(L, a, b)` returns the list $[[a..a + b, n_1], [a + b..a + 2b, n_2], \dots]$ where each number in L is in one of the intervals $[a + kb, a + (k + 1)b)$ and n_k is how many numbers from L are in the corresponding interval.
- `classes(L, a, M)` returns a similar list, but instead of $[[a..a + b, n_1], [a + b..a + 2b, n_2], \dots]$, the intervals are determined by a and the list of midpoints M .
- `classes(L, I)` returns a similar list, but instead of $[[a..a + b, n_1], [a + b..a + 2b, n_2], \dots]$, the intervals are given by I . In this case, not every element of L is necessarily in an interval.

Examples.

– *Input:*

```
classes([0,0.5,1,1.5,2,2.5,3,3.5,4],0,2)
```

Output:

$$\begin{bmatrix} 0.0 \dots 2.0 & 4 \\ 2.0 \dots 4.0 & 4 \\ 4.0 \dots 6.0 & 1 \end{bmatrix}$$

– *Input:*

```
classes([0,0.5,1,1.5,2,2.5,3,3.5,4],-1,2)
```

Output:

$$\begin{bmatrix} -1.0 \dots 1.0 & 2 \\ 1.0 \dots 3.0 & 4 \\ 3.0 \dots 5.0 & 3 \end{bmatrix}$$

– *Input:*

```
classes([0,0.5,1,1.5,2,2.5,3,3.5,4],1,[1,3,5])
```

Output:

$$\begin{bmatrix} 0.0 \dots 2.0 & 4 \\ 2.0 \dots 4.0 & 4 \\ 4.0 \dots 6.0 & 1 \end{bmatrix}$$

– *Input:*

```
classes([0,0.5,1,1.5,2,2.5,3,3.5,4],[1..3,3..6])
```

Output:

$$\begin{bmatrix} 1\dots 3 & 4 \\ 3\dots 6 & 3 \end{bmatrix}$$

9.1.10 Histograms: `histogram` `histogramme`

Given a list of intervals and a number of points in each interval, such as is given by the output of the `classes` command (see Section 9.1.9 p.718), a histogram is a graph consisting of a box over each interval, where the height of each box is proportional to the number of points and the total area of the boxes is 1. The `histogram` command draws histograms. The data can be sorted or unsorted.

`histogramme` is a synonym for `histogram`.

With sorted data:

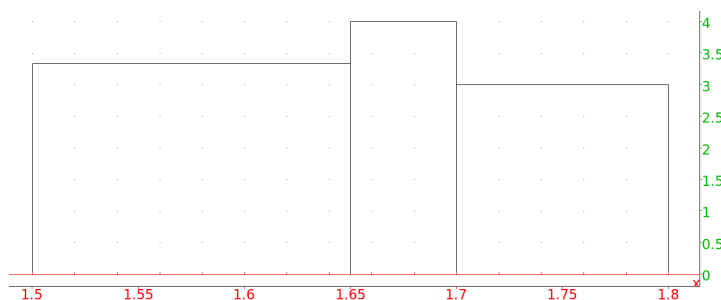
- `histogram` takes one argument:
 L , a list whose elements lists of a range $a..b$ and a positive integer.
- `histogram(L)` draws a histogram for the data.

Example.

Input:

```
histogram([[1.5..1.65,50],[1.65..1.7,20],[1.7..1.8,30]])
```

Output:

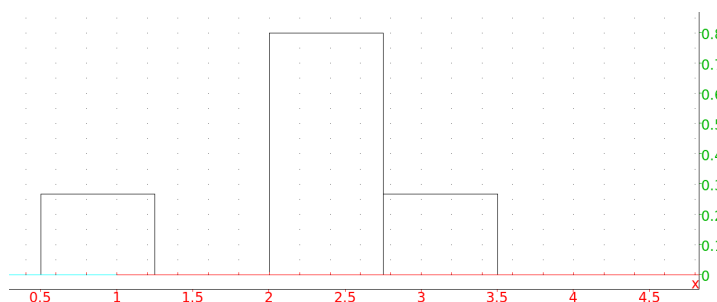


With unsorted data:

- `histogram` takes one mandatory argument and two optional arguments.
 - * L , a list of numbers.
 - * Optionally, a and b , numbers.
- `histogram(L⟨a,b⟩)` returns the histogram for `classes(L⟨a,b⟩)` (see Section 9.1.9 p.718).

Example.*Input:*

```
histogram([1,2,2.5,2.5,3],0.5,0.75)
```

Output:**9.1.11 Accumulating terms: `accumulate_head_tail`**

The `accumulate_head_tail` command replaces the first terms of a list by their sum and the last terms of a list by their sum.

- `accumulate_head_tail` takes three arguments:
 - * L , a list of numbers.
 - * n , the number of initial terms to add.
 - * m , the number of end terms to add.
- `accumulate_head_tail(L, n, m)` returns the list with the n initial terms and m end terms replaced by their sums.

Example.*Input:*

```
accumulate_head_tail([1,2,3,4,5,6,7,8,9,10],3,4)
```

Output:

```
[6, 4, 5, 6, 34]
```

9.1.12 Frequencies: `frequencies` `frequencies`

The frequency of a number in a list is the fraction of the list equal to the number. The `frequencies` command finds the frequencies of the numbers in a list.

`frequencies` is a synonym for `frequencies`.

- `frequencies` takes one argument:
 - L , a list of numbers.
- `frequencies(L)` returns a list whose elements are the numbers in the list and their frequencies.

Example.*Input:*

```
frequencies([1,2,1,1,2,1,2,4,3,3])
```

Output:

$$\begin{bmatrix} 1 & 0.4 \\ 2 & 0.3 \\ 3 & 0.2 \\ 4 & 0.1 \end{bmatrix}$$

You can use this, for example, to simulate flipping a fair coin and seeing how many times each side appears; to flip a coin 1000 times, for example:

Input:

```
frequencies([rand(2) $ (k=1..1000)])
```

Output (for example):

$$\begin{bmatrix} 0 & 0.484 \\ 1 & 0.516 \end{bmatrix}$$

(See Section 9.3.1 p.739 for information on `rand`.)

9.1.13 Cumulative frequencies: `cumulated_frequencies` `frequencies_cumulees`

Given a list of numbers L , the cumulated frequency at x is the fraction of numbers in the list less than x . The `cumulated_frequencies` command plots the cumulated frequency of the numbers in a list or given by a matrix.

For numbers in a list:

- `cumulated_frequencies` takes one argument:
 L , a list of numbers.
- `cumulated_frequencies(L)` draws the cumulated frequency of the numbers in L , where if L is a matrix, each number in the first column is repeated the number of times given in the second column.

For numbers in a matrix:

- `cumulated_frequencies` takes one argument:
 M , a matrix.
- `cumulated_frequencies(M)` (for a matrix with two columns, whose first column consists of numbers and whose second column consists of positive integers) draws the cumulated frequency of the numbers in the first column, where number in the first column is repeated the number of times given in the second column.

- `cumulated_frequencies(M)` (for a matrix with more than two columns, whose first column consists of numbers and whose remaining columns consist of positive integers) draws the cumulated frequencies for the first column paired with each remaining column.
- `cumulated_frequencies(M)` (for a matrix with two columns, whose first column consists of ranges $a..b$ and whose second column consists of positive numbers), will normalize the second column so the elements add up to 1 and draw the cumulated frequencies where the second column gives the frequency for the intervals in the first column.

Examples.

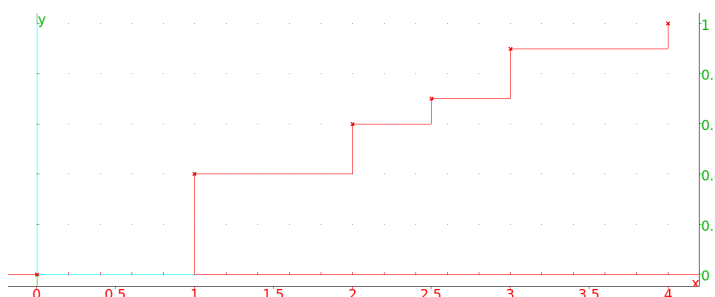
– *Input:*

```
cumulated_frequencies([1,2,1,1,2,1,2,4,3,3])
```

or:

```
cumulated_frequencies([[1,4],[2,3],[3,2],[4,1]])
```

Output:



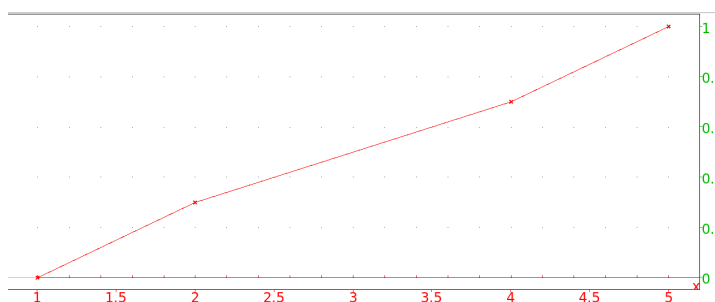
– *Input:*

```
cumulated_frequencies([[1..2,30],[2..4,40],[4..5,30]])
```

or:

```
cumulated_frequencies([[1..2,0.3],[2..4,0.4],[4..5,0.3]])
```

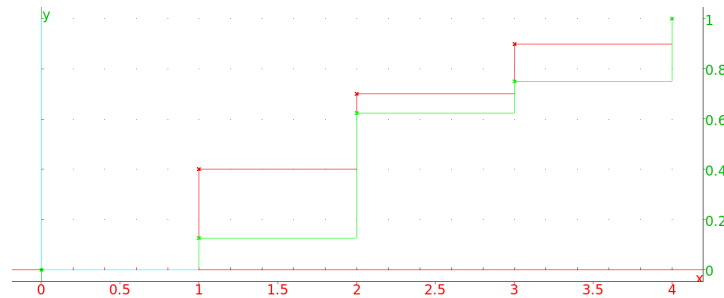
Output:



– *Input:*

```
cumulated_frequencies([[1,4,1],[2,3,4],[3,2,1],
                      [4,1,2]])
```

Output:



Here, both the distributions given by $[[1,4],[2,3],[3,2],[4,1]]$ and $[[1,1],[2,4],[3,1],[4,2]]$ are drawn on the same axes.

9.1.14 Bar graphs: `bar_plot`

The `bar_plot` command draws bar graphs.

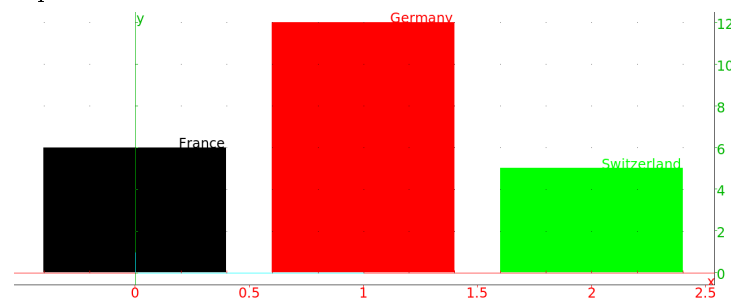
- * `bar_plot` takes one argument:
 M , a matrix, where each row consists of a label followed by one or more values. If the labels are followed by more than one value, then the first row needs to be identifiers.
- * `bar_plot(M)` draws a bar graph with a bar for each label, whose height is given by the corresponding value. If the matrix has more than two columns, then there will be a bar graph for each column of values.

Examples.

* *Input:*

```
bar_plot(["France", 6], ["Germany", 12],
         ["Switzerland", 5])
```

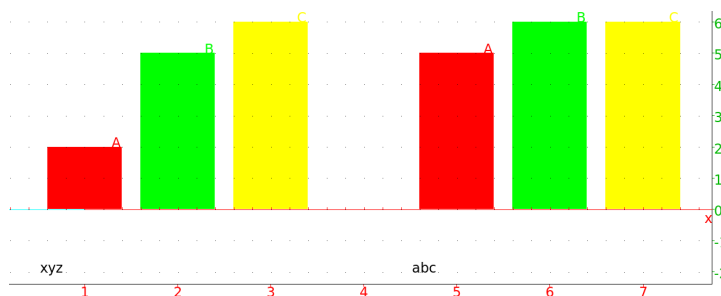
Output:



* *Input:*

```
bar_plot([["xyz", "abc"], ["A", 2, 5], ["B", 5, 6], ["C", 6, 6]])
```

Output:



9.1.15 Pie charts: `camembert`

You can draw pie charts using the same structure as bar graphs. The `camembert` command draws pie charts.

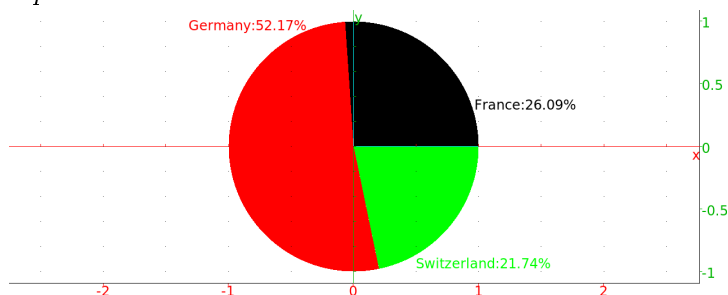
- * `camembert` takes one argument:
 M , a matrix, where each row consists of a label followed by one or more values. If the labels are followed by more than one value, then the first row needs to be identifiers.
- * `camembert(M)` draws a pie chart with a sector for each label, whose size is determined by the corresponding value. If the matrix has more than two columns, then there will be a pie chart for each column of values.

Examples.

* *Input:*

```
camembert(["France", 6], ["Germany", 12],
          ["Switzerland", 5])
```

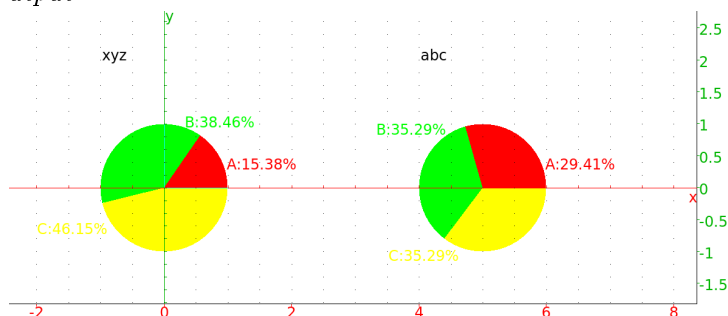
Output:



* *Input:*

```
camembert([2, "xyz", "abc"], ["A", 2, 5], ["B", 5, 6], ["C", 6, 6])
```

Output:



9.2 Two variable statistics

9.2.1 Covariance and correlation: `covariance` `covariance_correlation`

The covariance of two random variables measures their connectedness; i.e., whether they tend to change with each other. If X and Y are two random variables, then the covariance is the expected value of $(X - \bar{X})(Y - \bar{Y})$, where \bar{X} and \bar{Y} are the means of X and Y , respectively. The `covariance` command calculates covariances.

- * `covariance` takes two mandatory and one optional argument:
 - X and Y , two lists.
 - Optionally, W , a list of weights or a matrix (w_{jk}) where w_{jk} is the weight of the pair (x_j, y_k) .

If the arguments are all lists, then can be entered as the columns of a single matrix.

If the arguments consist of two lists and a matrix, to make it simpler to enter the data in a spreadsheet the lists X and Y and the matrix W can be combined into a single matrix, by augmenting W with the list Y on the top and the transpose of the list X on the left, with a filler in the upper left hand corner:

$$\begin{pmatrix} "XY" & Y \\ X^T & W \end{pmatrix}$$

For this, you have to give `covariance` a second argument of -1.

- `covariance(X, Y, W)` returns the covariance of X and Y .

Examples.

- *Input:*

```
covariance([1,2,3,4],[1,4,9,16])
```

or:

```
covariance([[1,1],[2,4],[3,9],[4,16]])
```

Output:

$$\frac{25}{4}$$

- *Input:*

```
covariance([1,2,3,4],[1,4,9,16],[3,1,5,2])
```

or:

```
covariance([1,2,3,4],[1,4,9,16],[[3,0,0,0],[0,1,0,0],[0,0,5,0],[0,0,0,2]])
```

or:

```
covariance(["XY",1,4,9,16],[1,3,0,5,0],[2,0,1,0,0],[3,0,0,5,0],[4,0,0,
```

Output:

$$\frac{662}{121}$$

```
covariance(["XY",
1,4,9,16],[1,3,0,5,0],[2,0,1,0,0],[3,0,0,5,0],[4,0,0,0,2]],-1)
```

The linear correlation coefficient of two random variables is another way to measure their connectedness. Given random variables X and Y , their correlation is defined as $\text{cov}(X,Y)/(\sigma(X)\sigma(Y))$, $\text{cov}(X,Y)$ is the covariance of X and Y , and $\sigma(X)$ and $\sigma(Y)$ are the standard deviations of X and Y , respectively.

The **correlation** command finds the correlation of two lists and take the same types of arguments as the **covariance** command.

- **correlation** takes two mandatory and one optional argument:
 - * X and Y , two lists.
 - * Optionally, W , a list of weights or a matrix (w_{jk}) where w_{jk} is the weight of the pair (x_j, y_k) .

If the arguments are all lists, then can be entered as the columns of a single matrix.

If the arguments consist of two lists and a matrix, to make it simpler to enter the data in a spreadsheet the lists X and Y and the matrix W can be combined into a single matrix, by augmenting W with the list Y on the top and the transpose of the list X on the left, with a filler in the upper left hand corner:

$$\begin{pmatrix} "XY" & Y \\ X^T & W \end{pmatrix}$$

For this, you have to give **correlation** a second argument of -1.

- **correlation**($X, Y \langle, W \rangle$) returns the correlation of X and Y .

Example.

Input:

```
correlation([1,2,3,4],[1,4,9,16])
```

Output:

$$\frac{100}{4\sqrt{645}}$$

The **covariance_correlation** command will compute both the covariance and correlation simultaneously, and return a list with both values. This command takes the same type of arguments as the **covariance** and **correlation** commands.

- `covariance_correlation` takes two mandatory and one optional argument:
 - X and Y , two lists.
 - Optionally, W , a list of weights or a matrix (w_{jk}) where w_{jk} is the weight of the pair (x_j, y_k) .

If the arguments are all lists, then can be entered as the columns of a single matrix.

If the arguments consist of two lists and a matrix, to make it simpler to enter the data in a spreadsheet the lists X and Y and the matrix W can be combined into a single matrix, by augmenting W with the list Y on the top and the transpose of the list X on the left, with a filler in the upper left hand corner:

$$\begin{pmatrix} "XY" & Y \\ X^T & W \end{pmatrix}$$

For this, you have to give `covariance_correlation` a second argument of -1.

- `covariance_correlation(X, Y, W)` returns a list consisting of the covariance and the correlation of X and Y .

Example.

Input:

```
covariance_correlation([1,2,3,4],[1,4,9,16])
```

Output:

$$\left[\frac{25}{4}, \frac{100}{4\sqrt{645}} \right]$$

9.2.2 Scatterplots: `scatterplot` `nuaged_points` `batons`

A scatter plot is simply a set of points plotted on axes. The `scatterplot` command draws scatter plots.

`nuage_points` is a synonym for `scatterplot`.

- `scatterplot` takes two arguments:
 - $xcoords$ and $ycoords$, a list of x -coordinates and y -coordinates. You can also combine them into a matrix with two columns (each list becomes a column of the matrix).
- `scatterplot(xcoords, ycoords)` draws the points with the given coordinates.

Example.

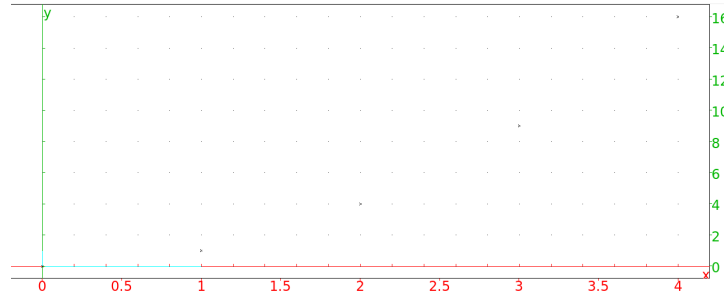
Input:

```
scatterplot([[0,0],[1,1],[2,4],[3,9],[4,16]])
```

or:

```
scatterplot([0,1,2,3,4],[0,1,4,9,16])
```

Output:



The **batons** command will also draw a collection of points, but each point will be connected to the x -axis with a vertical line segment.

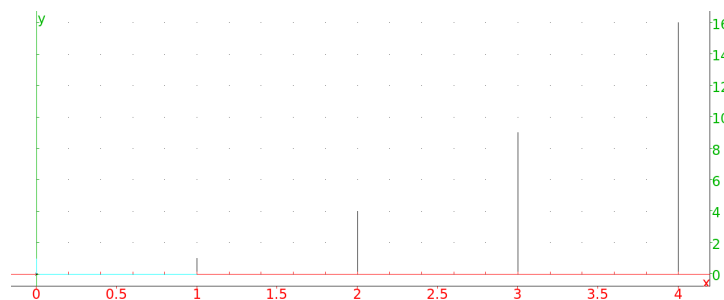
- **batons** takes two arguments:
 $xcoords$ and $ycoords$, a list of x -coordinates and y -coordinates. You can also combine them into a matrix with two columns (each list becomes a column of the matrix).
- **batons**($xcoords, ycoords$) draws the points with the given coordinates and connects them to the x -axis with vertical line segments.

Example.

Input:

```
batons([[0,0],[1,1],[2,4],[3,9],[4,16]])
```

Output:



9.2.3 Polygonal paths: polygonplot ligne_polygonale linear_interpolate listplot plotlist

The **polygonplot** command draws a polygonal path through given points. **polygonscatterplot** is a synonym for **polygonplot**.

- **polygonplot** takes one mandatory argument and one optional argument:
 - Optionally, $xcoords$, a list of x -coordinates. By default, the x -coordinates will be a list of integers starting at 0.

- `ycoords`, a list of y -coordinates.

You can combine two arguments into a matrix with two columns (each list becomes a column of the matrix).

- `polygonplot($\langle xcoords \rangle$ $ycoords$)` draws the polygonal path through the given points, from left to right (so the points are automatically ordered by increasing x -coordinate).

Examples.

- *Input:*

```
polygonplot([0,1,2,3,4],[0,1,4,9,16])
```

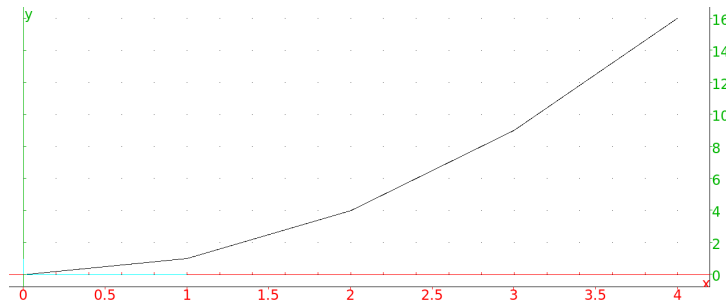
or:

```
polygonplot([[0,0],[1,1],[2,4],[3,9],[4,16]])
```

or:

```
polygonplot([[2,4],[0,0],[3,9],[1,1],[4,16]])
```

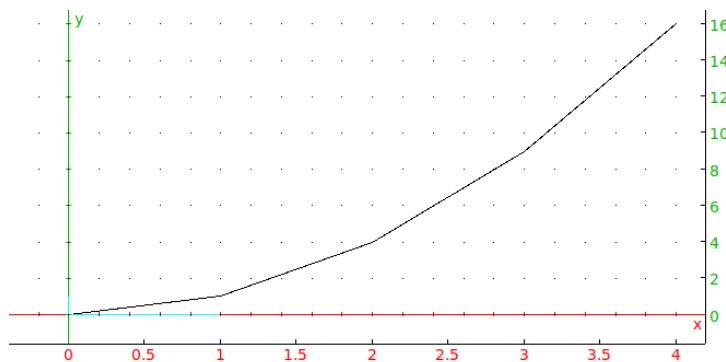
Output:



- *Input:*

```
polygonplot([0,1,4,9,16])
```

Output:



The `listplot` draws a polygonal path, but in an order determined by you.

`plotlist` is a synonym for `listplot`.

- `plotlist` takes one argument:
 L , a list of points or a list of numbers (which will be taken as y -coordinates, with the x -coordinates being the integers starting at 0).
- `plotlist(L)` draws a polygonal path through the points in the order given by the list.

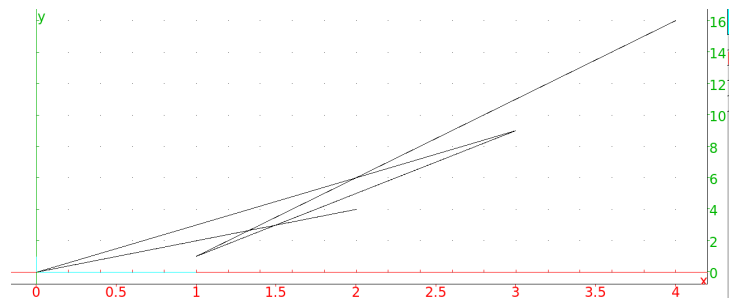
Unlike `polygonplot`, the `listplot` command can't be given two lists of numbers as arguments.

Examples.

- *Input:*

```
listplot([[2,4],[0,0],[3,9],[1,1],[4,16]])
```

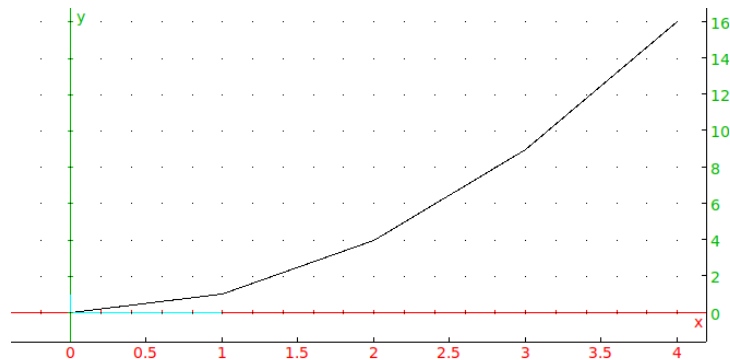
Output:



- *Input:*

```
listplot([0,1,4,9,16])
```

Output:



If you want to get coordinates on the polygonal path, you can use the `linear_interpolate` command. The `linear_interpolate` command will find coordinates on the polygonal path.

- `linear_interpolate` takes four arguments:
 - M , a two-row matrix consisting of the x -coordinates and the y -coordinates.
 - x_{min} , the minimum value of x that you are interested in.
 - x_{max} , the maximum value of x .
 - x_{step} , the step size that you want.

The values of x_{min} and x_{max} must be between the smallest and largest x -coordinates of the points.

- `linear_interpolate($M, x_{min}, x_{max}, x_{step}$)` returns a matrix with two rows, the first row will be $[x_{min}, x_{min} + x_{step}, x_{min} + 2x_{step}, \dots, x_{max}]$ and the second row will be the corresponding y -coordinates of the points on the polygonal path.

Example.

Input:

```
linear_interpolate([[1,2,6,9],[3,4,6,12]],2,7,1)
```

Output:

$$\begin{bmatrix} 2.0 & 3.0 & 4.0 & 5.0 & 6.0 & 7.0 \\ 4.0 & 4.5 & 5.0 & 5.5 & 6.0 & 8.0 \end{bmatrix}$$

9.2.4 Linear regression: `linear_regression` `linear_regression_plot`

Given a set of points $(x_0, y_0), \dots, (x_{n-1}, y_{n-1})$, linear regression finds the line $y = mx + b$ that comes closest to passing through all of the points; i.e., that makes

$$\sqrt{(y_0 - (mx_0 + b))^2 + \dots + (y_{n-1} - (mx_{n-1} + b))^2}$$

as small as possible. The `linear_regression` command finds the linear regression of a set of points.

- `linear_regression` takes two arguments:
 - $xcoords$, a list of x -coordinates.
 - $ycoords$, a list of y -coordinates.

You can combine two arguments into a matrix with two columns (each list becomes a column of the matrix).

- `linear_regression($xcoords, ycoords$)` returns a sequence m, b of the slope and y -intercept of the regression line.

Example.

Input:

```
linear_regression([[0,0],[1,1],[2,4],[3,9],[4,16]])
```

or:

```
linear_regression([0,1,2,3,4],[0,1,4,9,16])
```

Output:

4, -2

which means that the line $y = 4x - 2$ is the best fit line.

The `linear_regression_plot` command draws the best fit line.

- `linear_regression_plot` takes two arguments:

- *xcoords*, a list of x -coordinates.
- *ycoords*, a list of y -coordinates.

You can combine two arguments into a matrix with two columns (each list becomes a column of the matrix).

- `linear_regression_plot(xcoords,ycoords)` draws the line of best fit through the points. It will also give you the equation at the top, as well as the R^2 value, which is

$$R^2 = \frac{\sum_{j=0}^{n-1} (mx_j + b - \bar{y})^2}{\sum_{j=0}^{n-1} (y_j - \bar{y})^2}$$

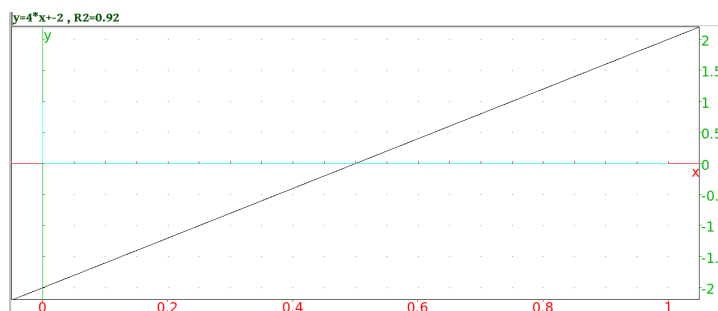
(The R^2 value will be between 0 and 1 and is one measure of how good the line fits the data; a value close to 1 indicates a good fit, a value close to 0 indicates a bad fit.)

Example.

Input:

```
linear_regression_plot([0,1,2,3,4],[0,1,4,9,16])
```

Output:



9.2.5 Exponential regression: `exponential_regression` `exponential_regression_plot`

You might expect a set of points to lie on an exponential curve $y = ba^x$. The `exponential_regression` command finds the values of a and b which give you the best fit exponential.

- `exponential_regression` takes two arguments:

- $xcoords$, a list of x -coordinates.
- $ycoords$, a list of y -coordinates.

You can combine two arguments into a matrix with two columns (each list becomes a column of the matrix).

- `exponential_regression($xcoords, ycoords$)` returns a sequence a, b of the numbers in the best fit exponential $y = ba^x$.

Example.

Input:

```
evalf(exponential_regression([[1,1],[2,4],[3,9],[4,16]]))
```

or:

```
evalf(exponential_regression([1,2,3,4],[1,4,9,16]))
```

(where the `evalf` is used to get a numeric approximation to an exact expression, see Section 6.8.1 p.171).

Output:

2.49146187923, 0.5

so the best fit exponential curve will be $y = 0.5 * (2.49146187923)^x$.

The `exponential_regression_plot` command draws the best fit exponential.

- `exponential_regression_plot` takes two arguments:

- $xcoords$, a list of x -coordinates.
- $ycoords$, a list of y -coordinates.

You can combine two arguments into a matrix with two columns (each list becomes a column of the matrix).

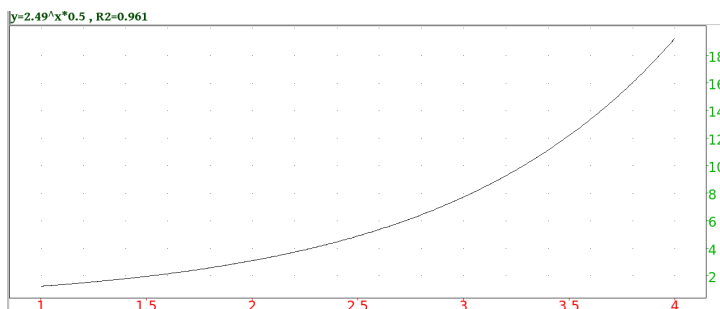
- `exponential_regression_plot($xcoords, ycoords$)` draws the best fit exponential, and puts the equation and R^2 value above the graph.

Example.

Input:

```
exponential_regression_plot([1,2,3,4],[1,4,9,16])
```

Output:



9.2.6 Logarithmic regression: `logarithmic_regression`

You might expect a set of points to lie on a logarithmic curve $y = m \ln(x) + b$. The `logarithmic_regression` command finds the logarithmic curve of best fit.

- `logarithmic_regression` takes two arguments:
 - *xcoords*, a list of x -coordinates.
 - *ycoords*, a list of y -coordinates.

You can combine two arguments into a matrix with two columns (each list becomes a column of the matrix).

- `logarithmic_regression(xcoords,ycoords)` returns a sequence m, b of the numbers in the best fit logarithmic curve $y = m \ln(x) + b$.

Example.

Input:

```
evalf(logarithmic_regression([[1,1],[2,4],[3,9],[4,16]]))
```

or:

```
evalf(logarithmic_regression([1,2,3,4],[1,4,9,16]))
```

(where the `evalf` is used to get a numeric approximation to an exact expression):

Output:

```
10.1506450002, -0.564824055818
```

so the best fit logarithmic curve will be $y = 10.1506450002 \ln(x) - 0.564824055818$.

The `logarithmic_regression_plot` command draws the best fit logarithmic curve.

- `logarithmic_regression_plot` takes two arguments:
 - *xcoords*, a list of x -coordinates.
 - *ycoords*, a list of y -coordinates.

You can combine two arguments into a matrix with two columns (each list becomes a column of the matrix).

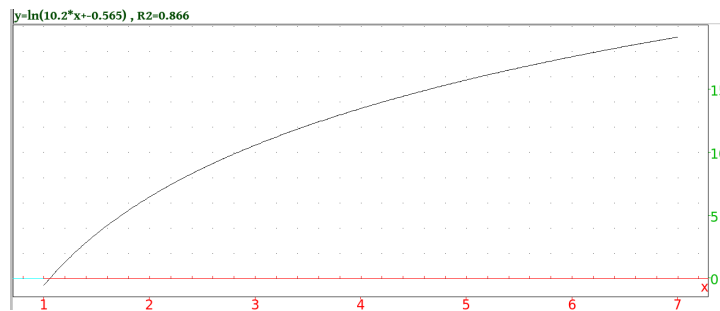
- `logarithmic_regression_plot(xcoords,ycoords)` draws the best fit logarithmic curve, and puts the equation and R^2 value above the graph.

Example.

Input:

```
logarithmic_regression_plot([1,2,3,4],[1,4,9,16])
```

Output:



9.2.7 Power regression: `power_regression` `power_regression_plot`

The `power_regression` command finds the graph $y = bx^m$ which best fits a set of data points.

- `power_regression` takes two arguments:
 - `xcoords`, a list of x -coordinates.
 - `ycoords`, a list of y -coordinates.

You can combine two arguments into a matrix with two columns (each list becomes a column of the matrix).

- `power_regression(xcoords,ycoords)` returns a sequence m, b of the numbers in the best fit power equation $y = bx^m$.

Example.

Input:

```
power_regression([[1,1],[2,4],[3,9],[4,16]])
```

or:

```
power_regression([1,2,3,4],[1,4,9,16])
```

Output:

2.0, 1.0

so the best fit (in this case, exact fit) power curve will be $y = 1.0x^2$.

The `power_regression_plot` command draws the best fit power function.

- `power_regression_plot` takes two arguments:

- *xcoords*, a list of x -coordinates.
- *ycoords*, a list of y -coordinates.

You can combine two arguments into a matrix with two columns (each list becomes a column of the matrix).

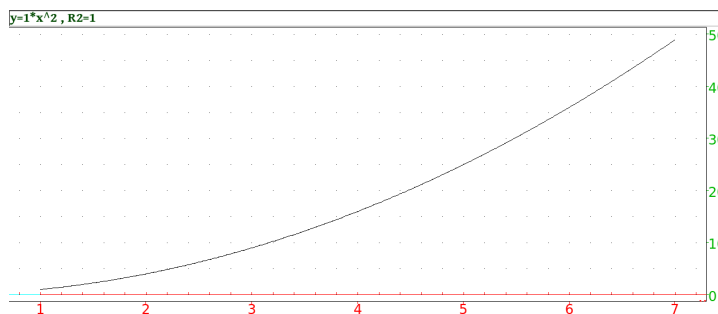
- `power_regression_plot(xcoords,ycoords)` draws the best fit power function, and puts the equation and R^2 value above the graph.

Example.

Input:

```
power_regression_plot([1,2,3,4],[1,4,9,16])
```

Output:



Note that in this case the R^2 value is 1, indicating that the data points fall directly on the curve.

9.2.8 Polynomial regression: `polynomial_regression`

The `polynomial_regression` command finds a more general polynomial $y = a_0x^n + \dots + a_n$ which best fits a set of data points.

- `polynomial_regression` takes three arguments:
 - *xcoords*, a list of x -coordinates.
 - *ycoords*, a list of y -coordinates.
 - n , the degree of the polynomial.

You can combine the first two arguments into a matrix with two columns (each list becomes a column of the matrix).

- `polynomial_regression(xcoords,ycoords,n)` returns the list $[a_n, \dots, a_0]$ of coefficients of the best fit polynomial.

Example.

Input:

```
polynomial_regression([[1,1],[2,2],[3,10],[4,20]],3)
```

or:


```
polynomial_regression([1,2,3,4],[1,2,10,20],3)
```

Output:

$$\left[-\frac{5}{6}, \frac{17}{2}, -\frac{56}{3}, 12\right]$$

so the best fit polynomial will be $y = (-5/6)x^3 + (17/2)x^2 - (56/3)x + 12$.

The `polynomial_regression_plot` command draws the best fit polynomial.

- `polynomial_regression_plot` takes three arguments:

- `xcoords`, a list of x -coordinates.
- `ycoords`, a list of y -coordinates.
- `n`, the degree of the polynomial.

You can combine the first two arguments into a matrix with two columns (each list becomes a column of the matrix).

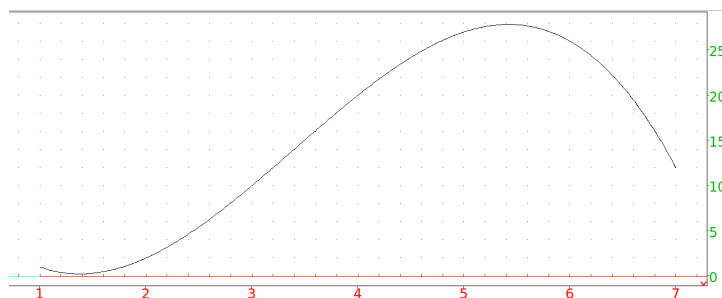
- `polynomial_regression_plot(xcoords,ycoords,n)` draws the best fit polynomial of degree n , and puts the equation and R^2 value above the graph.

Example.

Input:

```
polynomial_regression_plot([1,2,3,4],[1,2,10,20],3)
```

Output:



9.2.9 Logistic regression: `logistic_regression` `logistic_regression_plot`

Differential equations of the form $y' = y(a * y + b)$ come up often, particularly when studying bounded population growth. With the initial condition $y(x_0) = y_0$, the solution is the logistic equation

$$y = \frac{-b * y_0}{a * y_0 - (a * y_0 + b) \exp(b(x_0 - x))}$$

However, you often don't know the values of a and b . You can approximate these values given (x_0, y_0) and $[y'(x_0), y'(x_0+1), \dots, y'(x_0+n-1)]$ by taking

the initial value $y(x_0) = y_0$ and the approximation $y(t+1) \approx y(t) + y'(t)$ to get the approximations

$$\begin{aligned} y(x_0 + 1) &\approx y_0 + y'(x_0) \\ y(x_0 + 2) &\approx y_0 + y'(x_0) + y'(x_0 + 1) \\ &\vdots \\ y(x_0 + n) &\approx y_0 + y'(x_0) + \dots y'(x_0 + n - 1) \end{aligned}$$

Since $y'/y = a + by$, you can take the approximate values of $y'(x_0 + j)/y(x_0 + j)$ and use linear interpolation to get the best fit values of a and b , and then solve the differential equation.

The `logistic_regression` command uses this approach to find the best fit logistic equation for given data.

- `logistic_regression` takes three arguments:
 - L , a list representing $[y_{10}, y_{11}, \dots, y_{1(n-1)}]$, where y_{1j} represents the value of $y'(x_0 + j)$.
 - x_0 , the initial x value.
 - y_0 , the initial y value.
- `logistic_regression(L, x_0, y_0)` returns a list $[y, y', C, y_{max}, x_{max}, R, Y]$ where y is the logistic function, y' is the derivative, $C = -b/a$, y_{max} is the maximum value of y' , x_{max} is where y' has its maximum, R is linear correlation coefficient R of $Y = y'/y$ as a function of y with $Y = a * y + b$.

Example.

Input:

```
logistic_regression([0.0,1.0,2.5],0,1)
```

Output:

```
Pinstant=0.132478632479*Pcumul+0.0206552706553
Correlation 0.780548607383, Estimated total P=-0.155913978495
Returning estimated Pcumul, Pinstant, Ptotal, Pinstantmax, tmax, R
```

$$\left[\begin{aligned} & - \frac{0.155913978495}{1 + e^{-0.0554152581707x + (0.140088513344 + 3.14159265359i)}}, \\ & \frac{0.00161022271237}{1 + \cos(-i(-0.0554152581707x + (0.140088513344 + 3.14159265359i)))}, \\ & -0.155913978495, -0.000805111356186, 2.52797727501 + 56.6918346552i, \\ & 0.780548607383 \end{aligned} \right]$$

The `logistic_regression_plot` command draws the best fit logistic equation.

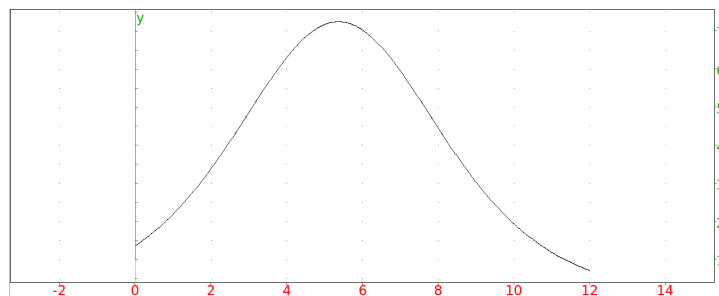
- `logistic_regression_plot` takes three arguments:
 - L , a list representing $[y_{10}, y_{11}, \dots, y_{1(n-1)}]$, where y_{1j} represents the value of $y'(x_0 + j)$.
 - x_0 , the initial x value.
 - y_0 , the initial y value.
- `logistic_regression_plot(L, x_0, y_0)` draws the best fit logistic equation.

Example.

Input:

```
logistic_regression_plot([1,2,4,6,8,7,5],0,2.0)
```

Output:



9.3 Random numbers

9.3.1 Producing uniformly distributed random numbers: `rand` and `random alea hasard sample`

The `rand` command produces random numbers, chooses random elements from a list, or creates functions that produce random numbers.

`rand` and `hasard` are synonyms for `rand`.

To produce random real numbers:

- `rand` takes two optional arguments.
 - Optionally, a and b , two real numbers. By default, $a = 0$ and $b = 1$.
- `rand([a,b])` returns a number in $[a,b)$ randomly and with equal probability.

Examples.

- *Input:*

```
rand()
```

(to produce a random number in $[0,1)$). **Output (for example):**

```
0.528489416465
```

- *Input:*

```
rand(1,1.5)
```

(to produce a random number in $[1, 1.5)$). *Output (for example):*

```
1.0012010464
```

To produce random integers:

- **rand** takes one argument:
 n , an integer.
- **rand(n)** returns a random integer in $[0, n)$ (or $(n, 0]$ if n is negative).

Example.

Input:

```
rand(5)
```

Output (for example):

```
3
```

You can then use **rand** to find a random integer in a specified interval; if you want an random integer between 6 and 10, inclusive, for example, you can enter:

Input:

```
6 + rand(11-6)
```

Output (for example):

```
7
```

Another way to get a random integer in a specified interval is with the **randint** command.

- **randint** takes two arguments:
 n_1 and n_2 , two integers.
- **randint(n_1, n_2)** returns a random integer between n_1 and n_2 , inclusive.

Example.

Input:

```
randint(6,10)
```

Output (for example):

```
8
```

To make a function which produces random numbers:

- **rand** takes one argument:
 $a..b$, a range with real numbers a and b .

- `rand(a..b)` returns a function which will generate a random number in the interval from a to b .

Example.

Input:

```
r:=rand(1.0..2.5);
r()
```

Output (for example):

1.68151313369

To choose elements without replacement:

- `rand` takes two or three arguments:
 - p , a positive integer.
 - Either: n_1 and n_2 , two integers.
 - or: L , a list.
- `rand(p, n_1, n_2)` returns a list of p distinct random integers from n_1 to n_2 .
- `rand(L)` returns p elements without replacement from the list L .

Examples.

- *Input:*

```
rand(2,1,10)
```

Output (for example):

[2, 9]

- *Input:*

```
rand(3,["a","b","c","d","e","f","g","h"])
```

Output (for example):

["e", "g", "a"]

- The list can have repeated elements.

Input:

```
rand(4,["r","r","r","r","v","v","v"])
```

Output (for example):

["r", "v", "v", "r"]

The `sample` command will also randomly select items from a list without replacement.

- `sample` takes two arguments:
 - L , a list.
 - p , an integer.
- `sample(L, p)` returns a list of p items chosen randomly from L , without replacement.

Note that with the `sample` command, the list comes first and then the integer.

Example.

Input:

```
sample(["r","r","r","r","v","v","v"],4)
```

Output (for example):

```
["v","v","v","r"]
```

9.3.2 Initializing the random number generator: `srand` `randseed` `RandSeed`

The `srand` and `RandSeed` commands initialize (or re-initialize) the random numbers given by `rand`.

`randseed` is a synonym for `srand`.

- `srand` takes one optional argument:
Optionally, n , an integer.
- `srand(n)` initializes the random numbers.
- `srand` (no parentheses) initializes the random numbers using the system clock.
- `RandSeed` takes one argument:
 n , an integer.
- `RandSeed(n)` initializes the random numbers.

9.3.3 Producing random numbers with the binomial distribution: `randbinomial`

The `randbinomial` command finds random numbers chosen according to the binomial distribution (see Section 9.4.3 p.761).

- `randbinomial` takes two arguments:
 - n , an integer.
 - p , a probability (a number between 0 and 1).
- `randbinomial(n, p)` returns an integer from 0 to n chosen randomly according to the binomial distribution with parameters n and p ; i.e., the number of successes you might get if you did an experiment n times, where the probability of success each time is p .

Example.*Input:*`randbinomial(100,0.4)`*Output:*

42

9.3.4 Producing random numbers with a multinomial distribution: randmultinomial

The `randmultinomial` command finds random numbers chosen according to a multinomial distribution (see Section 9.4.5 p.765).

- `randmultinomial` takes one mandatory and one optional argument:
 - P , a list $P = [p_0, \dots, p_{n-1}]$ of n probabilities which add to 1 (representing the probability that one of several mutually exclusive events occurs).
 - Optionally, K , a list of length n .
- `randmultinomial(L)` returns an index chosen randomly according to the corresponding multinomial distribution.
- `randmultinomial(L, K)` returns an element of K whose index is chosen randomly.

Examples.

- *Input:*

`randmultinomial([1/2, 1/3, 1/6])`*Output (for example):*

1

- *Input:*

`randmultinomial([1/2, 1/3, 1/6],["R","V","B"])`*Output (for example):*

"R"

9.3.5 Producing random numbers with a Poisson distribution: randpoisson

Recall that given a number $\lambda > 0$, the corresponding Poisson distribution $P(\lambda)$ satisfies

$$\text{Prob}(X \leq k) = \exp(-\lambda) \lambda^k / k!$$

It will have mean λ and standard deviation $\sqrt{\lambda}$. (See also Section 9.4.6 p.766.)

The `randpoisson` command finds a random integer according to a Poisson distribution.

- `randpoisson` takes one argument:
 λ , a positive number.
- `randpoisson(λ)` returns an integer chosen randomly according to the Poisson distribution with parameter λ .

Example.*Input:*

```
randpoisson(10.6)
```

Output (for example):

```
16
```

9.3.6 Producing random numbers with a normal distribution: `randnorm` `randNorm`

The `randnorm` command chooses a random number according to a normal distribution.

`randNorm` is a synonym for `randnorm`.

- *randnorm* takes two arguments:
 - μ , a real number (the mean).
 - σ , a positive real number (the standard deviation).
- `randnorm(μ, σ)` returns a number chosen randomly according to the normal distribution with mean μ and standard deviation σ .

Example.*Input:*

```
randnorm(2,1)
```

Output (for example):

```
3.39283224858
```

9.3.7 Producing random numbers with Student's distribution: `randstudent`

The `randstudent` command finds random numbers chosen according to Student's distribution (see Section 9.4.8 p.770).

- `randstudent` takes one argument:
 n , an integer (the degrees of freedom).
- `randstudent(n)` returns a number chosen randomly according to Student's distribution with n degrees of freedom.

Example.*Input:*

```
randstudent(5)
```

Output (for example):

```
0.268225314184
```


9.3.8 Producing random numbers with the χ^2 distribution: `randchisquare`

The `randchisquare` command finds random numbers chosen according to the χ^2 distribution (see Section 9.4.9 p.773).

- `randchisquare` takes one argument:
 n , an integer (the degrees of freedom).
- `randchisquare(n)` returns a number chosen randomly according to the χ^2 distribution with n degrees of freedom.

Example.

Input:

```
randchisquare(5)
```

Output (for example):

```
4.53970828547
```

9.3.9 Producing random numbers with the Fisher-Snédécór distribution: `randfisher`

The `randfisher` command finds random numbers chosen according to the Fisher-Snédécór distribution (see Section 9.4.10 p.775).

- `randfisher` takes two arguments:
 n_1 and n_2 , integers (degrees of freedom).
- `randfisher(n_1, n_2)` returns a number chosen randomly according to the Fisher-Snédécór distribution with n_1 and n_2 degrees of freedom.

Example.

Input:

```
randfisher(2,3)
```

Output (for example):

```
2.33137725333
```

9.3.10 Producing random numbers with the gamma distribution: `randgammad`

The `randgammad` command finds random numbers chosen according to the gamma distribution (see Section 9.4.11 p.777).

- `randgammad` takes two arguments:
 a and b , positive real numbers (the parameters).
- `randgammad(a, b)` returns a number chosen randomly according to the gamma distribution with parameters a and b .

Example.

Input:

```
randgammad(3,1)
```

Output (for example):

```
4.91461463472
```

9.3.11 Producing random numbers with the beta distribution: `randbetad`

The `randbetad` command finds random numbers chosen according to the beta distribution (see Section 9.4.12 p.779).

- `randbetad` takes two arguments:
 a and b , positive real numbers (the parameters).
- `randbetad(a, b)` returns a number chosen randomly according to the beta distribution with parameters a and b .

Example.

Input:

```
randbetad(2,3)
```

Output (for example):

```
0.524453873081
```

9.3.12 Producing random numbers with the geometric distribution: `randgeometric`

The `randgeometric` command finds random numbers chosen according to the geometric distribution (see Section 9.4.13 p.780).

- `randgeometric` takes one argument:
 p , a probability (a number between 0 and 1).
- `randgeometric(p)` returns a number chosen randomly according to the geometric distribution with probability p .

Example.

Input:

```
randgeometric(0.2)
```

Output (for example):

```
11
```

9.3.13 Producing random numbers with the exponential distribution: `randexp`

The `randexp` command finds random numbers chosen according to the exponential distribution (see Section 9.4.15 p.784).

- `randexp` takes one argument:
 λ , a positive real number (the parameter).

- **randexp**(λ) returns a number chosen randomly according to the exponential distribution with parameter λ .

Example.

Input:

```
randexp(2.1)
```

Output (for example):

```
0.0288626239833
```

9.3.14 Random variables: `random_variable` `randvar`

The **randvar** command produces an object representing a random variable. The value(s) can be generated subsequently by calling **sample** (see Section 9.3.1 p.739), **rand** (see Section 9.3.1 p.739), **randvector** (see Section 9.3.15 p.756) or **randmatrix** (see Section 9.3.16 p.757).

random_variable is a synonym for **randvar**.

- **randvar** takes a sequence of arguments:
distspec, which can specify a probability distribution. The distributions and their arguments are:
 - Uniform distribution (see Section 9.4.2 p.759):
 Arguments:
 - * **uniform** or **uniformd**.
 - * Either:
 - Optionally, **mean**= μ , to specify a mean of μ .
 - Optionally, **stddev**= σ , to specify a standard deviation of σ .
 - Optionally, **variance**= σ^2 , to specify a variance of σ^2 .
 - or:
 - a and b , two numbers specifying the end points of a range.
 The range can also be specified by $a..b$ or **range**= $a..b$.
 - Binomial distribution (see Section 9.4.3 p.761):
 Arguments:
 - * **binomial**.
 - * Either:
 - n , a positive integer.
 - p , a probability (a number between 0 and 1).
 - or:
 - Optionally, **mean**= μ , to specify a mean of μ .
 - Optionally, **stddev**= σ , to specify a standard deviation of σ .
 - Optionally, **variance**= σ^2 , to specify a variance of σ^2 .
 - Negative binomial distribution (see Section 9.4.4 p.763):
 Arguments:

- * `negbinomial`.
- * n , a positive integer.
- * p , a probability (a number between 0 and 1).
- Multinomial distribution (see Section 9.3.4 p.743):
Arguments:
 - * `randmultinomial`.
 - * $[p_0, p_1, \dots, p_j]$, a list of probabilities with $p_0 + \dots + p_j = 1$.
 - * Optionally, $[a_0, a_1, \dots, a_j]$, a list of possible return values.
- Poisson distribution (see Section 9.4.6 p.766):
Arguments:
 - * `poisson`.
 - * λ , a real number
or
`mean= μ` , the mean value.
- Standard normal distribution (see Section 9.3.6 p.744):
Argument:
 - * `randnorm` or `randNorm`.
- Normal distribution (see Section 9.4.7 p.767):
Arguments:
 - * `normald`.
 - * Either no arguments (for the standard normal distribution)
or one of:
 - Optionally, `mean= μ` , to specify a mean of μ .
 - Optionally, `stddev= σ` , to specify a standard deviation of σ .
 - Optionally, `variance= σ^2` , to specify a variance of σ^2 .
 - or:
 - a and b , two numbers specifying the end points of a range.
The range can also be specified by `a..b` or `range=a..b`.
- Student's distribution (see Section 9.4.8 p.770):
Arguments:
 - * `student`.
 - * n , an integer (the degrees of freedom).
- χ^2 distribution (see Section 9.4.9 p.773):
Arguments:
 - * `chisquare`.
 - * n , an integer (the degrees of freedom).
- Fisher-Snedécor distribution (see Section 9.4.10 p.775):
Arguments:
 - * `fisher`, `fisherd`, or `snedecor`.
 - * n_1 and n_2 , integers (the degrees of freedom).

- Gamma distribution (see Section 9.4.11 p.777):
Arguments:
 - * `gammad`.
 - * a and b , real numbers.
 - Beta distribution (see Section 9.4.12 p.779):
Arguments:
 - * `betaa`.
 - * a and b , real numbers.
 - Geometric distribution (see Section 9.4.13 p.780):
Arguments:
 - * `geometric`.
 - * p , a probability (number between 0 and 1).
 - Cauchy distribution (see Section 9.4.14 p.782):
Arguments:
 - * `cauchy` or `cauchyd`.
 - * a and b , real numbers.
 - Exponential distribution (see Section 9.4.15 p.784):
Arguments:
 - * `exponential` or `exponentiald`.
 - * λ , a real number.
 - Weibull's distribution (see Section 9.4.16 p.785):
Arguments:
 - * k , an integer.
 - * λ , a real number.
 - Discrete distribution:
Arguments:
 - * $W = [w_1, w_2, \dots, w_n]$, a list of nonnegative weights.
 - * Optionally, $V = [v_1, v_2, \dots, v_n]$, a list of values.
- or:
- * $[[v_1, w_1], [v_2, w_2], \dots, [v_n, w_n]]$, a list of object-weight pairs.
- or:
- * f , a nonnegative function.
 - * $a..b$ or `range=a..b` with real numbers a and b , a range specification.
 - * Optionally, N , a positive integer or $V = [v_0, v_1, v_2, \dots, v_n]$, a list of values with $n = b - a$ (here a and b have to be integers).

The weights are automatically scaled by the inverse of their sum to obtain the values of the probability mass function. If a function f is given instead of a list of weights, then $w_k = f(a + k)$ for $k = 0, 1, \dots, b - a$ unless N is given, in which case $w_k = f(x_k)$ where $x_k = a + (k - 1) \frac{b - a}{N}$ and $k = 1, 2, \dots, N$. The resulting random variable X has values in $\{0, 1, \dots, n - 1\}$ for 0-based modes

(e.g. Xcas) resp. in $\{1, 2, \dots, n\}$ for 1-based modes (e.g. Maple).
 If the list V of custom objects is given, then $V[X]$ is returned instead of X . If N is given, then $v_k = x_k$ for $k = 1, 2, \dots, N$.

- `randvar(distspec)` returns an object representing a random variable.

Examples.

- Define a random variable with a Fisher-Snedecor distribution (two degrees of freedom):

Input:

```
X:=random_variable(fisher,2,3)
```

Output:

```
fisherd(2,3)
```

To generate some values of X :

Input:

```
rand(X)
```

or:

```
sample(X)
```

Output:

```
0.30584514472
```

Input:

```
randvector(5,X)
```

or:

```
sample(X,5)
```

Output:

```
[2.2652, 0.1397, 6.3320, 1.0556, 0.2995]
```

- Define a random variable with multinomial distribution:

Input:

```
M:=randvar(multinomial,[1/2,1/3,1/6],[a,b,c])
```

Output:

```
multinomial, [1/2, 1/3, 1/6], [a, b, c]
```

Input:

```
randvector(10,M)
```

Output:

$[b, b, b, b, b, b, a, a, b, b]$

Some continuous distributions can be defined by specifying their first and/or second moment.

Examples.

- *Input :*

`randvector(10,randvar(poisson,mean=5))`

Output:

$[7, 2, 5, 6, 7, 9, 8, 4, 3, 4]$

- *Input:*

`randvector(5,randvar(weibull,mean=5.0,stddev=1.5))`

Output:

$[1.6124, 3.2720, 7.02627, 5.5360, 3.1929]$

- *Input:*

`X:=randvar(binomial,mean=18,stddev=4)`

Output:

$\binom{162}{\frac{1}{9}}$

- *Input:*

`X:=randvar(weibull,mean=12.5,variance=1)`

Output:

`weibulld(3.08574940721,13.9803128143)`

Input:

`mean(randvector(1000,X))`

Output:

12.5728578447

- *Input:*

`G:=randvar(geometric,stddev=2.5)`

Output:

`geometric(0.327921561087)`

Input:

```
evalf(stddev(randvector(1000,G)))
```

Output:

```
2.57913473863
```

- *Input:*

```
randvar(gammad,mean=12,variance=4)
```

Output:

```
gammad(36,3)
```

Uniformly distributed random variables can be defined by specifying the support as an interval.

Examples:

- *Input:*

```
randvector(5,randvar(uniform,range=15..81))
```

Output:

```
[77.0025, 77.7644, 63.2414, 52.0707, 66.3837]
```

- *Input:*

```
rand(randvar(uniform,e..pi))
```

Output:

```
3.1010453504
```

The following examples demonstrate various ways to define a discrete random variable.

Examples:

- *Input:*

```
X:=randvar(["apple",1/3],["orange",1/4], ["pear",1/5],["plum",13/60]);
randvector(5,X)
```

Output (for example):

```
["orange", "apple", "apple", "plum", "apple"]
```

- *Input:*

```
W:= [1,4,5,3,1,1,1,2]:: X:=randvar(W)::
approx(W/sum(W))
```


Output (for example):

[0.0556, 0.2222, 0.2778, 0.1667, 0.0556, 0.0556, 0.0556, 0.1111]

- *Input:*

```
frequencies(randvector(10000,X))
```

(See Section 9.1.12 p.720.)

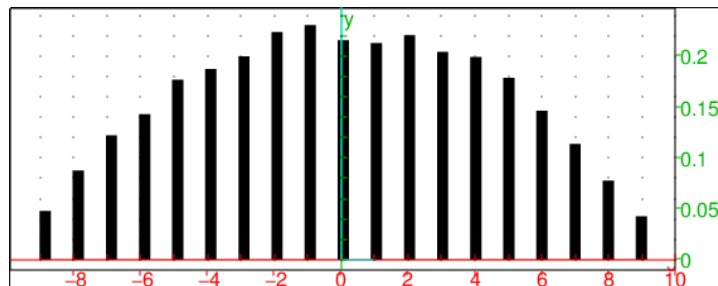
Output:

0	0.0527
1	0.2189
2	0.2791
3	0.1698
4	0.0546
5	0.0557
6	0.059
7	0.1102

- *Input:*

```
X:=randvar(k->1-(k/10)^2,range=-10..10)::  
histogram(randvector(10000,X),-10,0.33,display=filled)
```

Output:



- *Input:*

```
X:=randvar([3,1,2,5],[alpha,beta,gamma,delta]):; randmatrix(5,4,X)
```

Output:

$$\begin{pmatrix} \delta & \delta & \beta & \delta \\ \delta & \gamma & \gamma & \beta \\ \alpha & \delta & \alpha & \delta \\ \alpha & \alpha & \gamma & \alpha \\ \delta & \delta & \beta & \delta \end{pmatrix}$$

Discrete random variables can be used to approximate custom continuous random variables. For example, consider a probability density function f as a mixture of two normal distributions on the support $S = [-10, 10]$. You can sample f in $N = 10000$ points in S .

Input:

```

F:=normald(3,2,x)+normald(-5,1,x)::
c:=integrate(F,x=-10..10)::
f:=unapply(1/c*F,x)::
X:=randvar(f,range=-10..10,10000)::

```

Now generate 25000 values of X and plot a histogram:

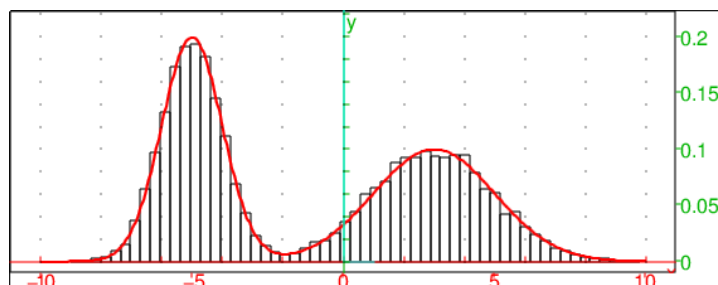
Input:

```

R:=sample(X,25000)::
hist:=histogram(R,-10,0.1)::
PDF:=plot(f(x),display=red+line_width_2)::
hist,PDF

```

Output:



Sampling from discrete distributions is fast: for instance, generating 25 million samples from the distribution of X which has about 10000 outcomes takes only couple of seconds. In fact, the sampling complexity is constant. Also, observe that the process isn't slowed down by spreading it across multiple calls of `randvector`.

Input:

```

for k from 1 to 1000 do randvector(25000,X); od::

```

Evaluation time: 2.12

Independent random variables can be combined in an expression, yielding a new random variable. In the example below, you define a log-normally distributed variable Y from a variable X with standard normal distribution.

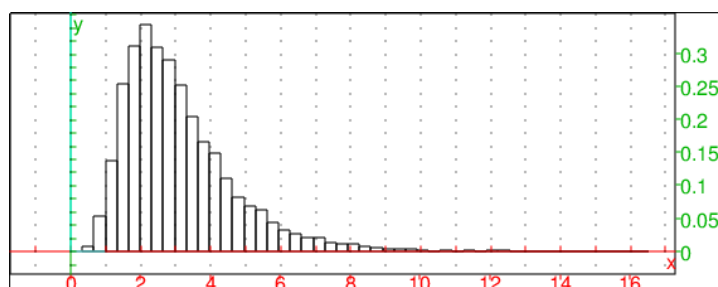
Input:

```

X:=randvar(normal):: mu,sigma:=1.0,0.5::
Y:=exp(mu+sigma*X)::
L:=randvector(10000,Y)::
histogram(L,0,0.33)

```

Output:



It is known that $E[Y] = e^{\mu+\sigma^2/2}$. The mean of L should be close to that number.

Input:

```
mean(L); exp(mu+sigma^2/2)
```

Output:

```
3.0789, 3.0802
```

In case a compound random variable is defined as an expression containing several independent random variables X, Y, \dots of the same type, you sometimes need to prevent its evaluation when passing it to **randvector** and similar functions.

Example.

Input:

```
X:=randvar(normal):: Y:=randvar(normal)::
```

If you want to generate, for example, the random variable X/Y , you would have to forbid automatic evaluation of the latter expression; otherwise it would reduce to 1 since X and Y are both **normald**(0,1).

Input:

```
randvector(5, eval(X/Y, 0))
```

Output (for example):

```
[-0.358479277895, 5.03004946974, -5.5414073892, -0.885656967277, -2.63689662108]
```

To save typing, you can define Z with **eval**(*,0) and pass **eval**(Z ,1) to **randvector** or **randmatrix**.

Input:

```
Z:=eval(X/Y, 0):: randvector(5, eval(Z, 1))
```

Output (for example):

```
[0.404123429613, -4.06194898981, 0.00356038536404, 1.61619003525, -2.85682173195]
```

Parameters of a distribution can be entered as symbols to allow (re)assigning them at any time.

Input:

```
purge(lambda)::  
X:=randvar(exp, lambda)::  
lambda:=1::
```

Now execute the following command line several times in a row. The parameter λ is updated in each iteration.

Input:

```
r:=rand(X); lambda:=sqrt(r)
```

Output (by executing the above command line three times):

```
8.5682, 2.9272  
1.5702, 1.2531  
0.53244, 0.72968
```

9.3.15 Make a random vector or list: `randvector`

The `randvector` command creates random vectors (see also Section 6.27.28 p.369).

- `randvector` takes one mandatory argument and one optional argument:
 - n , an integer.
 - Optionally, X , which can be an integer or a random variable. In place of a random variable, the specifications for a distribution can be used. (See Section 9.3.14 p.747 for random variables and their specifications.)
- `randvector(n, X)` returns a vector of size n containing random integers:
 - with no second argument, distributed uniformly between -99 and +99.
 - with a second argument of X , distributed uniformly between 0 and $k - 1$.
 - with a second argument the specification of a distribution, distributed according to this distribution.

Examples.

- *Input:*

```
randvector(3)
```

Output (for example):

```
[-64, -30, 70]
```

- *Input:*

```
randvector(3, 5)
```

or:

```
randvector(3, 'rand(5)')
```

Output (for example):

```
[2, 4, 2]
```

- *Input:*

```
randvector(3, 'randnorm(0,1)')
```

Output:

```
[-0.361127118455, -0.018325111754, 1.11875485898]
```

- *Input:*

```
randvector(3,2..4)
```

Output:

```
[3.18034843914, 2.48592940345, 2.57507958449]
```

9.3.16 Producing random matrices: `randmatrix` `ranm` `randMat`

The `randmatrix` command produces random vectors and matrices. (See also Section 6.27.28 p.369 and Section 9.3.15 p.756.)

`ranm` and `randMat` are synonyms for `randmatrix`.

- `randmatrix` takes one mandatory argument and three optional arguments:
 - n , an integer.
 - Optionally, p , an integer.
 - Optionally, a , an integer.
 - Optionally, $a..b$, a range.
 - Optionally, *distr*, a distribution, which can be one of:
 - * `'rand(n)'` (see Section 9.3.1 p.739).
 - * `'binomial(n,p)'`, `'binomial, n,p '` or `'randbinomial(n,p)'`, for a binomial distribution (see Section 9.4.3 p.761 and Section 9.3.3 p.742).
 - * `'multinomial(P,K)'`, `'multinomial, P,K '` or `'randmultinomial(P,K)'` for a multinomial distribution (see Section 9.4.5 p.765 and Section 9.3.4 p.743).
 - * `'poisson(λ)'`, `'poisson, λ '` or `'randpoisson(λ)'` for a Poisson distribution (see Section 9.4.6 p.766 and Section 9.3.5 p.743).
 - * `'normald(μ,σ)'`, `'normald, μ,σ '` or `'randnorm(μ,σ)'` for a normal distribution (see Section 9.4.7 p.767 and Section 9.3.6 p.744).
 - * `'exp(a)'`, `'exp, a '` or `'randexp(a)'` for an exponential distribution (see Section 9.4.15 p.784 and Section 9.3.13 p.746).
 - * `'fisher(n,m)'`, `'fisher, n,m '` or `'randfisher(n,m)'` for a Fisher-Snédécór distribution (see Section 9.4.10 p.775 and Section 9.3.9 p.745).

Note that *distr* is in quotes.

- `randmatrix(n)` returns a vector of length n whose elements are integers chosen randomly from $[-99, -98, \dots, 98, 99]$ with equal probability.

- `randmatrix(n, p)` returns an $n \times p$ matrix whose elements are integers chosen randomly from $[-99, 99]$ with equal probability.
- `randmatrix(n, p, a)` returns an $n \times p$ matrix whose elements are integers chosen randomly from $[0, a]$ (or $(a, 0]$ if a is negative) with equal probability.
- `randmatrix($n, p, a..b$)` returns an $n \times p$ matrix whose elements are real numbers chosen randomly from $[a, b]$ with equal probability.
- `randmatrix($n, p, distr$)` returns an $n \times p$ matrix whose elements are numbers chosen randomly according to distribution *distr*.

Examples.

- *Input:*

```
randmatrix(5)
```

Output (for example):

```
[-48, 54, 28, -51, 63]
```

- *Input:*

```
randmatrix(2,3)
```

Output (for example):

$$\begin{pmatrix} 40 & -74 & -87 \\ 40 & -19 & 20 \end{pmatrix}$$

- *Input:*

```
randmatrix(2,3,10)
```

Output (for example):

$$\begin{pmatrix} 4 & 2 & 1 \\ 4 & 4 & 0 \end{pmatrix}$$

- *Input:*

```
randmatrix(2,3,0..1)
```

Output (for example):

$$\begin{pmatrix} 0.384355471935 & 0.655490326229 & 0.924850208685 \\ 0.159429819323 & 0.952957109548 & 0.220945354551 \end{pmatrix}$$

- *Input:*

```
randmatrix(2,3,'randnorm(2,1)')
```

Output (for example):

$$\begin{pmatrix} 2.17670501195 & 0.653882567048 & 2.94543112983 \\ 2.46150672679 & 2.19251320854 & 2.44211638655 \end{pmatrix}$$

9.4 Density and distribution functions

9.4.1 Distributions and inverse distributions

Let $p(x)$ be a probability density function, so $p(x) \geq 0$ for all x , and for a discrete density function,

$$\sum_{x \in \mathbb{Z}} p(x) = 1$$

while for a continuous density function,

$$\int_{-\infty}^{\infty} p(x) = 1$$

The corresponding cumulative distribution function

$$P(x) = \text{Prob}(X \leq x)$$

is the probability that a randomly (according to the probability being considered) chosen value is less than or equal to x . This can be used to find the probability that a randomly chosen value is between two numbers:

$$\text{Prob}(x < X \leq y) = P(y) - P(x)$$

Given a value h between 0 and 1, the inverse distribution function for a distribution takes h to the value of x for which $\text{Prob}(X \leq x) = h$.

9.4.2 The uniform distribution

The probability density function for the uniform distribution: `uniform`
`uniformd`

Given two values a and b with $a < b$, the uniform distribution on $[a, b]$ has density function $1/(b - a)$ for x in $[a, b]$. The `uniform` (or `uniformd`) command computes this density function.

- `uniform` (or `uniformd`) takes three arguments:
 - a and b , real numbers with $a < b$.
 - x , a real number.
- `uniform(a, b, x)` (or `uniformd(a, b, x)`) returns the value of the probability density function for the uniform distribution from a to b , namely $1/(b - a)$.

Example.

Input:

```
uniform(2.2, 3.5, 2.8)
```

Output:

```
0.769230769231
```

The cumulative distribution function for the uniform distribution:`uniform_cdf uniformd_cdf`

The `uniform_cdf` command finds the cumulative distribution function for the uniform distribution.

- `uniform_cdf` takes three mandatory arguments and one optional argument:
 - a and b , real numbers with $a < b$.
 - x , a real number.
 - Optionally y , a real number.
- `uniform_cdf(a, b, x)` returns the value of the cumulative distribution function for the uniform distribution from a to b , which in this case will be $(x - a)/(b - a)$.
- `uniform_cdf(a, b, x, y)` returns $\text{Prob}(x \leq X \leq y)$, which in this case will be $(y - x)/(b - a)$.

Examples.

- *Input:*

```
uniform_cdf(2,4,3.2)
```

Output:

0.6

- *Input:*

```
uniform_cdf(2,4,3,3.2)
```

Output:

0.1

The inverse distribution function for the uniform distribution:`uniform_icdf uniformd_icdf`

The `uniform_icdf` command computes the inverse distribution for the uniform distribution.

- `uniform_icdf` takes three arguments:
 - a and b , real numbers with $a < b$.
 - h , a real number between 0 and 1.
- `uniform_icdf(a, b, h)` returns the value of the inverse distribution function to the uniform distribution from a to b ; namely the value of x for which $h = \text{Prob}(X \leq x)$.

Example.

Input:

```
uniform_icdf(2,3,.6)
```

Output:

2.6

9.4.3 The binomial distribution

The probability density function for the binomial distribution:
`binomial`

If you perform an experiment n times where the probability of success each time is p , then the probability of exactly k successes is:

$$\text{binomial}(n, k, p) = \binom{n}{k} p^k (1 - p)^{n-k} \quad (9.1)$$

This determines the binomial distribution.

The `binomial` command computes the density function for the binomial distribution.

- *binomial* takes two mandatory arguments and one optional argument.
 - n , a positive integer.
 - k , a nonnegative integer less than or equal to n .
 - Optionally, p , a probability (a real number between 0 and 1).
- `binomial(n, k)` returns the binomial coefficient $\binom{n}{k}$ (see Section 6.6.2 p.159), same as `comb(n, k)`
- `binomial(n, k, p)` returns the probability given by (9.1).

Examples.

- *Input:*

```
binomial(10,2)
```

or:

```
comb(10,2)
```

Output:

```
45
```

- *Input:*

```
binomial(10,2,0.4)
```

Output:

```
0.120932352
```

The cumulative distribution function for the binomial distribution:
`binomial_cdf`

The `binomial_cdf` command computes the cumulative distribution function for the binomial distribution.

- *binomial_cdf* takes three mandatory arguments and one optional argument:
 - n , a positive integer.
 - p , a probability (a real number between 0 and 1).
 - x , a real number.
 - Optionally, y , a real number.
- `binomial_cdf(n, p, x)` returns

$$\text{Prob}(X \leq x) = \text{binomial}(n, 0, p) + \dots + \text{binomial}(n, \text{floor}(x), p)$$
- `binomial_cdf(n, p, x, y)` returns

$$\text{Prob}(x \leq X \leq y) = \text{binomial}(n, \text{ceil}(x), p) + \dots + \text{binomial}(n, \text{floor}(y), p)$$

Examples.

- *Input:*

```
binomial_cdf(4,0.5,2)
```

Output:

```
0.6875
```

- *Input:*

```
binomial_cdf(2,0.3,1,2)
```

Output:

```
0.51
```

The inverse distribution function for the binomial distribution:
`binomial_icdf`

The `binomial_icdf` command computes the inverse distribution function for the binomial distribution.

- *binomial_icdf* takes three mandatory arguments and one optional argument:
 - n , a positive integer.
 - p , a probability (a real number between 0 and 1).
 - h , a real number between 0 and 1.

- `binomial_icdf(n, p, h)` returns the value of the inverse distribution for the binomial distribution with n trials and probability p ; namely, the smallest value of x for which $\text{Prob}(X \leq x) \geq h$.

Example.*Input:*`binomial_icdf(4,0.5,0.9)`*Output:*

3

Note that `binomial_cdf(4,0.5,3)` = 0.9375, which is bigger than 0.9, while `binomial_cdf(4,0.5,2)` = 0.6875, which is smaller than 0.9.

9.4.4 The negative binomial distribution**The probability density function for the negative binomial distribution: `negbinomial`**

If you repeatedly perform an experiment with probability of success p , then, given an integer n , the probability of k failures that occur before you have n successes is given by the negative binomial distribution, which can be computed by

$$\binom{n+k-1}{k} p^n (1-p)^k. \quad (9.2)$$

The `negbinomial` command finds the density function for the negative binomial distribution.

- `negbinomial` takes three arguments:
 - n and k , integers.
 - p , a probability (a real number between 0 and 1).
- `negbinomial(n, k, p)` returns the value of the negative binomial distribution, given in (9.2).

Example.*Input:*`negbinomial(4,2,0.5)`*Output:*

0.15625

Note that

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \frac{n(n-1)\dots(n-k+1)}{k!}$$

The second formula makes sense even if n is negative, and you can write

$$\text{negbinomial}(n, k, p) = \binom{-n}{k} p^n (p-1)^k,$$

from which the name negative binomial distribution comes from. This also makes it simple to determine the mean $(n(1-p)/p)$ and variance $(n(1-p)/p^2)$. The negative binomial is also called the Pascal distribution (after Blaise Pascal) or the Pólya distribution (after George Pólya).

The cumulative distribution function for the negative binomial distribution: `negbinomial_cdf`

The `negbinomial_cdf` command finds the cumulative distribution function for the negative binomial distribution.

- `negbinomial_cdf` takes three mandatory arguments and two optional arguments:
 - n , an integer.
 - p , a probability (between 0 and 1).
 - x , a number.
 - Optionally, y , a number.
- `negbinomial_cdf(n, p, x)` returns

$$\text{Prob}(X \leq x) = \text{negbinomial}(n, 0, p) + \dots + \text{negbinomial}(n, \text{floor}(x), p).$$
- `negbinomial_cdf(n, p, x, y)` returns

$$\text{Prob}(x \leq X \leq y) = \text{negbinomial}(n, \text{ceil}(x), p) + \dots + \text{negbinomial}(n, \text{floor}(y), p)$$

Examples.

- *Input:*

```
negbinomial_cdf(4,0.5,2)
```

Output:

```
0.34375
```

- *Input:*

```
negbinomial_cdf(4,0.5,2,5)
```

Output:

```
0.40234375
```

The inverse distribution function for the negative binomial distribution: `negbinomial_icdf`

The `negbinomial_icdf` command gives the inverse distribution function for the negative binomial distribution.

- `negbinomial_icdf` takes three arguments:
 - n , a positive integer.
 - p , a probability (a real number between 0 and 1).
 - h , a real number between 0 and 1.

- `negbinomial_icdf(n, p, h)` returns the value of the inverse distribution for the negative binomial distribution with n and probability p ; namely, the smallest value of x for which $\text{Prob}(X \leq x) \geq h$.

Example.

Input:

```
negbinomial_icdf(4,0.5,0.9)
```

Output:

8

9.4.5 The multinomial probability function: `multinomial`

If X follows a multinomial probability distribution with $P = [p_0, p_1, \dots, p_j]$ (where $p_0 + \dots + p_j = 1$), then for $K = [k_0, \dots, k_j]$ with $k_0 + \dots + k_j = n$, the probability that $X = K$ is given by

$$\frac{n!}{k_0!k_1!\dots k_j!} (p_0^{k_0} p_1^{k_1} \dots p_j^{k_j}). \quad (9.3)$$

The `multinomial` command computes the density function for the multinomial distribution.

- *multinomial* takes three arguments:
 - n , an integer.
 - $P = [p_0, p_1, \dots, p_j]$, a probability vector (i.e., $p_k \geq 0$ for all k and $p_0 + \dots + p_j = 1$).
 - $K = [k_0, \dots, k_j]$, a list of integers with $k_0 + \dots + k_j = n$.
- `multinomial(n, P, K)` returns the probability that $X = K$, given in (9.3).

You will get an error if $k_0 + \dots + k_j$ is not equal to n , although you won't get one if $p_0 + \dots + p_j$ is not equal to 1.

Example.

Suppose you make 10 choices, where each choice is one of three items; the first has a 0.2 probability of being chosen, the second a 0.3 probability and the third a 0.5 probability. The probability that you end up with 3 of the first item, 2 of the second and 5 of the third will be:

Input:

```
multinomial(10,[0.2,0.3,0.5],[3,2,5])
```

Output:

0.0567

9.4.6 The Poisson distribution

The probability density function for the Poisson distribution: `poisson`

Recall that for the Poisson distribution with parameter λ , the probability of a non-negative integer k is $e^{-\lambda}\lambda^k/k!$. This distribution has mean λ and variance λ .

The `poisson` command gives the density function for the Poisson distribution.

- `poisson` takes two arguments:
 - λ , a real number.
 - k , a non-negative integer.
- `poisson(λ, k)` returns the value of the Poisson density function with parameter λ at x , namely $e^{-\lambda}\lambda^k/k!$.

Example.

Input:

```
poisson(10.0,9)
```

Output:

```
0.125110035721
```

The cumulative distribution function for the Poisson distribution:
`poisson_cdf`

The `poisson_cdf` command computes the cumulative distribution function for the Poisson distribution.

- `poisson_cdf` takes two arguments:
 - μ , a real number.
 - x , a real number.
 - Optionally, y , a real number.

- `poisson_cdf(μ, x)` returns

$$\text{Prob}(X \leq x) = \text{poisson}(\mu, 0) + \dots + \text{poisson}(\mu, \text{floor}(x))$$

for the Poisson distribution with parameter μ .

- `poisson_cdf(μ, x, y)` returns

$$\text{Prob}(x \leq X \leq y) = \text{poisson}(\mu, \text{ceil}(x)) + \dots + \text{poisson}(\mu, \text{floor}(y))$$

Examples.

- *Input:*

```
poisson_cdf(10.0,3)
```

Output:

```
0.0103360506759
```

- *Input:*

```
poisson_cdf(10.0,3,10)
```

Output:

```
0.580270354477
```

The inverse distribution function for the Poisson distribution: `poisson_icdf`

The `poisson_icdf` command finds the inverse distribution function for the Poisson distribution.

- `poisson_icdf` takes three arguments:
 - μ , a real number.
 - h , a real number between 0 and 1.
- `poisson_icdf(μ, h)` returns the value of the inverse distribution for the Poisson distribution with parameter μ ; namely, the value of x for which $\text{Prob}(X \leq x) = h$.

Example.

Input:

```
poisson_icdf(10.0,0.975)
```

Output:

```
17
```

9.4.7 Normal distributions

The probability density function for a normal distribution: `normald` `loi_normal`

The density function of the normal distribution with mean μ and standard deviation σ at the point x is

$$\text{normald}(\mu, \sigma, x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-(x-\mu)^2/2} \quad (9.4)$$

The `normald` (or `loi_normal`) command finds the value of this density function.

- `normald` (or `loi_normal`) command takes two optional arguments and one mandatory argument:

- Optionally, μ and σ , the mean and standard deviation. (By default, $\mu = 0$ and $\sigma = 1$, giving the standard normal distribution.)
- x , a real number.
- `normald`($[\mu, \sigma,] x$) returns the value of the normal density function with parameter μ and standard deviation σ at the value x , given in (9.4).

Examples.

- *Input:*

```
normald(2,1,3)
```

Output:

$$\frac{e^{-\frac{1}{2}}}{\sqrt{2\pi}}$$

- *Input:*

```
normald(2)
```

Output:

$$\frac{1}{\sqrt{2\pi e^2}}$$

The cumulative distribution function for normal distributions: `normal_cdf`
`normald_cdf`

The `normal_cdf` (or `normald_cdf`) command computes the cumulative distribution function for the normal distribution.

- `normal_cdf` (or `normald_cdf`) takes three optional arguments and one mandatory argument:
 - Optionally, μ and σ , the mean and standard deviation. (By default, $\mu = 0$ and $\sigma = 1$, giving the standard normal distribution.)
 - x , a real number.
 - Optionally, y , a real number.
- `normal_cdf`($[\mu, \sigma,] x$) returns $\text{Prob}(X \leq x)$ for the normal distribution with mean μ and standard deviation σ .
- `normal_cdf`($[\mu, \sigma,] x, y$) returns $\text{Prob}(x \leq X \leq y)$.

Examples.

- *Input:*

```
normal_cdf(1,2,1.96)
```


Output:

0.684386303484

- *Input:*

`normal_cdf(1,2.1,1.2)`

Output:

0.537937144066

- *Input:*

`normal_cdf(1,2.1,1.2,9)`

Output:

0.461993238584

The inverse distribution function for normal distributions: `normal_icdf`
`normald_icdf`

The `normal_icdf` (or `normald_icdf`) command computes the inverse distribution for the normal distribution.

- *normal_icdf* (or *normald_icdf*) takes two optional arguments and one mandatory argument:
 - Optionally, μ and σ , the mean and standard deviation. (By default, $\mu = 0$ and $\sigma = 1$, giving the standard normal distribution.)
 - h , a real number.
- `normal_icdf([μ , σ ,] h)` returns the inverse distribution for the normal distribution with mean μ and standard deviation σ ; namely, the value of x for which $\text{Prob}(X \leq x) = h$.

Examples.

- *Input:*

`normal_icdf(0.975)`

Output:

1.95996398454

- *Input:*

`normal_icdf(1,2,0.495)`

Output:

0.974933060984

The upper tail cumulative function for normal distributions: UTPN

The **UTPN** (the Upper Tail Probability - Normal distribution) computes $\text{Prob}(X > x)$ for a normal distribution.

- *UTPN* takes two optional arguments and one mandatory argument:
 - Optionally, μ and σ^2 , the mean variance deviation. (By default, $\mu = 0$ and $\sigma^2 = 1$, giving the standard normal distribution.)
 - Note:** Unlike **normald** and **normal_cdf**, the **UTPN** takes the variance and not the standard deviation.
 - x , a real number.
- **UTPN**($[\mu, \sigma^2,] x$) returns $\text{Prob}(X > x)$, for the normal distribution with mean μ and variance σ^2 .

Examples.

- *Input:*

UTPN(1.96)

Output:

0.0249978951482

•

- *Input:*

UTPN(1,4,1.96)

Output:

0.315613696516

9.4.8 Student's distribution

The probability density function for Student's distribution: student studentd

Student's distribution (also called Student's t -distribution or just the t -distribution) with n degrees of freedom has density function given by

$$\text{student}(n, x) = \frac{\Gamma((n+1)/2)}{\Gamma(n/2)\sqrt{n\pi}} \left(1 + \frac{x^2}{n}\right)^{-n-1/2} \quad (9.5)$$

where recall the Gamma function (see Section 6.8.13 p.183) is defined for $x > 0$ by

$$\Gamma(x) = \int_0^\infty e^{-t} t^{x-1} dx.$$

The **student** command finds the density function for Student's distribution. **studentd** is a synonym for **student**.

- `student` takes two arguments:
 - n , an integer (the degrees of freedom).
 - x , a real number.

`student(n, x)` returns the value of the density function for Student's distribution with n degrees of freedom at x , given in (9.5).

Example.

Input:

```
student(2,3)
```

Output:

$$\frac{2\sqrt{\pi}\sqrt{\frac{11}{2}}^{-1}}{2\sqrt{2\pi} \cdot 11}$$

which can be numerically approximated by:

Input:

```
evalf(student(2,3))
```

Output:

```
0.0274101222343
```

The cumulative distribution function for Student's distribution:
`student_cdf`

The `student_cdf` command computes the cumulative distribution function for Student's distribution.

- `student_cdf` takes two mandatory arguments and one optional argument.
 - n , an integer (the degrees of freedom).
 - x , a real number.
 - Optionally, y , a real number.
- `student_cdf(n, x)` returns $\text{Prob}(X \leq x)$ for Student's distribution with n degrees of freedom.
- `student_cdf(n, x, y)` returns $\text{Prob}(x \leq X \leq y)$.

Examples.

- *Input:*

```
student_cdf(5,2)
```

Input:

```
0.949030260585
```

- *Input:*

```
student_cdf(5,-2,2)
```

Output:

```
0.89806052117
```

The inverse distribution function for Student's distribution: `student_icdf`

The `student_icdf` command computes the inverse distribution for Student's distribution.

- *student_icdf* takes two arguments:
 - n , an integer (the degrees of freedom).
 - h , a real number between 0 and 1.
- `student_icdf(n, h)` returns the inverse distribution for Student's distribution with n degrees of freedom; namely, the value of x for which $\text{Prob}(X \leq x) = h$.

Example.

Input:

```
student_icdf(5,0.95)
```

Output:

```
2.01504837333
```

The upper tail cumulative function for Student's distribution: `UTPT`

The `UTPT` (the Upper Tail Probability - T distribution) computes $\text{Prob}(X > x)$ for Student's distribution.

- *UTPT* takes two arguments:
 - n , an integer (the degrees of freedom).
 - x , a real number.
- `UTPT(n, x)` returns $\text{Prob}(X > x)$ for Student's distribution with n degrees of freedom.

Example.

Input:

```
UTPT(5,2)
```

Output:

```
0.0509697394149
```

9.4.9 The χ^2 distribution

The probability density function for the χ^2 distribution: `chisquare`

The χ^2 distribution with n degrees of freedom has density function given by

$$\chi^2(n, x) = \frac{x^{n/2-1} e^{-x/2}}{2^{n/2} \Gamma(n/2)} \quad (9.6)$$

The `chisquare` command computes this density function.

- `chisquare` takes two arguments:
 - n , an integer (the degrees of freedom).
 - x , a real number.
- `chisquare(n, x)` returns the value of the χ^2 density function with n degrees of freedom, given in (9.6).

Example.

Input:

```
chisquare(5,2)
```

Output:

$$\frac{2\sqrt{2}}{e\left(\frac{3}{4}\sqrt{\pi}\sqrt{2} \cdot 2^2\right)}$$

which can be numerically approximated by:

Input:

```
evalf(chisquare(5,2))
```

Output:

```
0.138369165807
```

The cumulative distribution function for the χ^2 distribution: `chisquare_cdf`

The `chisquare_cdf` command computes the cumulative distribution function for the χ^2 distribution.

- `chisquare_cdf` takes two mandatory arguments and one optional argument:
 - n , an integer (the degrees of freedom).
 - x , a real number.
 - Optionally, y , a real number.
- `chisquare_cdf(n, x)` returns $\text{Prob}(X \leq x)$ for the χ^2 distribution with n degrees of freedom.
- `chisquare_cdf(n, x, y)` returns $\text{Prob}(x \leq X < y)$.

Examples.

- *Input:*

```
chisquare_cdf(5,11)
```

Output:

```
0.948620016517
```

- *Input:*

```
chisquare_cdf(3,1,2)
```

Output:

```
0.22884525243
```

The inverse distribution function for the χ^2 distribution: `chisquare_icdf`

The `chisquare_icdf` command computes the inverse distribution for the χ^2 distribution.

- *chisquare_icdf* takes two arguments:
 - n , an integer (the degrees of freedom).
 - h , a real number between 0 and 1.
- `chisquare_icdf(n, h)` returns the inverse distribution for the χ^2 distribution with n degrees of freedom; namely, the value of x for which $\text{Prob}(X \leq x) = h$.

Example.

Input:

```
chisquare_icdf(5,0.95)
```

Output:

```
11.0704976935
```

The upper tail cumulative function for the χ^2 distribution: `UTPC`

The UTPC (the Upper Tail Probability - Chi-square distribution) computes $\text{Prob}(X > x)$ for the χ^2 distribution.

- *UTPC* takes two arguments:
 - n , an integer (the degrees of freedom).
 - x , a real number.
- `UTPC(n, x)` returns $\text{Prob}(X > x)$ for the χ^2 distribution with n degrees of freedom.

Example.

Input:

```
UTPC(5,11)
```

Output:

```
0.0513799834831
```

9.4.10 The Fisher-Snédécór distribution

The probability density function for the Fisher-Snédécór distribution: `fisher fisherd snedecor snedecord`

The Fisher-Snédécór distribution (also called the F-distribution) with n_1 and n_2 degrees of freedom has density function given by, for $x \geq 0$,

$$\text{fisher}(n_1, n_2, x) = \frac{(n_1/n_2)^{n_1/2} \Gamma((n_1 + n_2)/2)}{\Gamma(n_1/2) \Gamma(n_2/2)} \frac{x^{(n_1-2)/2}}{(1 + (n_1/n_2)x)^{(n_1+n_2)/2}} \quad (9.7)$$

The `fisher` command computes this density function.

`fisherd`, `snedecor` and `snedecord` are synonyms for `fisher`.

- `fisher` takes three arguments:
 - n_1 and n_2 , integers (the degrees of freedom).
 - x , a non-negative real number.
- `fisher(n_1, n_2, x)` returns the value of the Fisher-Snédécór density function with n_1 and n_2 degrees of freedom, given in (9.7).

Example.

Input:

```
fisher(5,3,2.5)
```

Output:

```
0.10131184472
```

The cumulative distribution function for the Fisher-Snédécór distribution: `fisher_cdf snedecor_cdf`

The `fisher_cdf` (or `snedecor_cdf`) command computes the cumulative distribution function for the Fisher-Snédécór distribution.

- `fisher_cdf` takes three mandatory arguments and one optional argument:
 - n_1 and n_2 , integers (the degrees of freedom).
 - x , a real number.
 - Optionally, y , a real number.
- `fisher_cdf(n_1, n_2, x)` returns $\text{Prob}(X \leq x)$ for the Fisher-Snédécór distribution with n_1 and n_2 degrees of freedom
- `fisher_cdf(n_1, n_2, x, y)` returns $\text{Prob}(x \leq X < y)$.

Examples.

- *Input:*

```
fisher_cdf(5,3,9)
```

Output:

$$\beta\left(\frac{5}{2}, \frac{3}{2}, \frac{15}{16}, 1\right)$$

(See Section 6.8.16 p.185.)

This can be numerically approximated with:

Input:

```
evalf(fisher_cdf(5,3,9))
```

Output:

```
0.949898927032
```

• *Input:*

```
evalf(fisher_cdf(5,3,9,10))
```

Output:

```
0.0066824173023
```

The inverse distribution function for the Fisher-Snédécór distribution: `fisher_icdf` `snedecor_icdf`

The `fisher_icdf` (or `snedecor_icdf`) command computes the inverse distribution for the Fisher-Snédécór distribution.

- *fisher_icdf* takes three arguments:
 - n_1 and n_2 , integers (the degrees of freedom).
 - h , a real number between 0 and 1.
- `fisher_icdf(n_1, n_2, h)` returns the inverse distribution for the Fisher-Snédécór distribution with n_1 and n_2 degrees of freedom; namely, the value of x for which $\text{Prob}(X \leq x) = h$.

Example.

Input:

```
fisher_icdf(5,3,0.95)
```

Output:

```
9.01345516752
```

The upper tail cumulative function for the Fisher-Snédécór distribution: `UTPF`

The `UTPF` (the Upper Tail Probability - Fisher-Snédécór distribution) computes $\text{Prob}(X > x)$ for the Fisher-Snédécór distribution.

- *UTPF* takes three arguments:
 - n_1 and n_2 , integers (the degrees of freedom).

- x , a real number.
- $\text{UTPF}(n_1, n_2, x)$ returns $\text{Prob}(X > x)$ for the Fisher-Snedécor distribution with n_1 and n_2 degrees of freedom.

Example.

Input:

`UTPF(5, 3, 9)`

Output:

0.050101072968

9.4.11 The gamma distribution

The probability density function for the gamma distribution: `gammad`

The gamma distribution depends on two parameters, $a > 0$ and $b > 0$; the value of the density function at $x \geq 0$ is

$$\text{gammad}(a, b, x) = x^{a-1} e^{-bx} b^a / \Gamma(a) \quad (9.8)$$

The `gammad` command computes this density function.

- `gammad` takes three arguments:
 - a and b , positive real numbers (the parameters).
 - x , a real number.
- `gammad(a, b, x)` returns the value of the gamma density function with parameters a and b , given in (9.8).

Example.

Input:

`gammad(2, 1, 3)`

Output:

$$\frac{3}{e^3}$$

The cumulative distribution function for the gamma distribution: `gammad_cdf`

The `gamma_cdf` command computes the cumulative distribution function for the gamma distribution.

- `gamma_cdf` takes three mandatory arguments and one optional argument:
 - a and b , real numbers (the parameters).
 - x , a real number.

- Optionally, y , a real number.
- `gamma_cdf(a, b, x)` returns $\text{Prob}(X \leq x)$ for the gamma distribution with parameters a and b .
- `gamma_cdf(n, x, y)` returns $\text{Prob}(x \leq X \leq y)$.

It turns out that

$$\text{gammad_cdf}(n, x) = \text{igamma}(a, bx, 1)$$

where `igamma` is the incomplete gamma function (see Section 6.8.15 p.184),

$$\text{igamma}(a, x, 1) = \int_0^x e^{-t} t^{a-1} dt / \Gamma(a).$$

Examples.

- *Input:*

`gammad_cdf(2, 1, 0.5)`

Output:

0.090204010431

- *Input:*

`gammad_cdf(2, 1, 0.5, 1.5)`

Output:

0.351970589198

The inverse distribution function for the gamma distribution: `gammad_icdf`

The `gammad_icdf` command computes the inverse distribution for the gamma distribution.

- `gamma_icdf` takes three arguments:
 - a and b , numbers (the parameters).
 - h , a real number between 0 and 1.
- `gamma_icdf(a, b, h)` returns the inverse distribution for the gamma distribution with parameters a and b ; namely, the value of x for which $\text{Prob}(X \leq x) = h$.

Example.

Input:

`gammad_icdf(2, 1, 0.5)`

Output:

1.67834699002

9.4.12 The beta distribution

The probability density function for the beta distribution: `betad`

The beta distribution depends on two parameters, $a > 0$ and $b > 0$; the value of the density function at x in $[0, 1]$ is

$$\text{betad}(a, b, x) = \Gamma(a + b)x^{a-1}(1 - x)^{b-1}/(\Gamma(a)\Gamma(b)) \quad (9.9)$$

(see Section 6.8.13 p.183).

The `betad` command computes the density function for the beta distribution.

- *betad* takes three arguments:
 - a and b , positive numbers, the parameters.
 - x , a real number.
- `betad(a, b, x)` returns the value of the density function for the beta distribution with parameters a and b , given in (9.9).

Example.

Input:

```
betad(2,1,0.3)
```

Output:

```
0.6
```

The cumulative distribution function for the beta distribution:
`betad_cdf`

The `betad_cdf` command computes the cumulative distribution function for the beta distribution.

- *betad_cdf* takes three mandatory arguments and one optional argument:
 - a and b , real numbers (the parameters).
 - x , a real number.
 - Optionally, y , a real number.
- `betad_cdf(a, b, x)` returns $\text{Prob}(X \leq x)$ for the beta distribution with parameters a and b .
- `betad_cdf(n, x, y)` returns $\text{Prob}(x \leq X \leq y)$.

It turns out that

$$\text{betad_cdf}(a, b, x) = \beta(a, b, x)\Gamma(a + b)/(\Gamma(a)\Gamma(b))$$

where $\beta(a, b, x) = \int_0^x t^{a-1}(1 - t)^{b-1}dt$ (see Section 6.8.16 p.185).

Examples.

- *Input:*

```
betad_cdf(2,3,0.2)
```

Output:

```
0.1808
```

- *Input:*

```
betad_cdf(2,3,0.25,0.5)
```

Output:

```
0.42578125
```

The inverse distribution function for the beta distribution: `betad_icdf`

The `betad_icdf` command computes the inverse distribution for the beta distribution.

- `betad_icdf` takes three arguments:
 - a and b , real numbers (the parameters).
 - h , a real number between 0 and 1.
- `betad_icdf(a, b, h)` returns the inverse distribution for the beta distribution with parameters a and b ; namely, the value of x for which $\text{Prob}(X \leq x) = h$.

Example.

Input:

```
betad_icdf(2,3,0.2)
```

Output:

```
0.212317128278
```

9.4.13 The geometric distribution

The probability density function for the geometric distribution: `geometric`

If an experiment with probability of success p is iterated, the probability that the first success occurs on the k th trial is $(1 - p)^{k-1}p$. This gives the geometric distribution (with parameter p) on the natural numbers. Given such a p , the geometric density function at n is given by

$$\text{geometric}(p, n) = (1 - p)^{n-1}p \quad (9.10)$$

The `geometric` command computes this density function.

- `geometric` takes two arguments:
 - p , a probability (a number between 0 and 1).

- x , a real number.

`geometric(p, x)` returns the value of the geometric density function with probability p , given in (9.10).

Example.

Input:

```
geometric(0.2,3)
```

Output:

```
0.128
```

The cumulative distribution function of the geometric distribution:
`geometric_cdf`

The `geometric_cdf` command computes the cumulative distribution function for the geometric distribution.

- *geometric_cdf* takes three mandatory arguments and one optional argument:
 - p , a probability (a number between 0 and 1).
 - n , a natural number.
 - Optionally, k , a natural number.
- `betad_cdf(p, n)` returns $\text{Prob}(X \leq n)$ for the geometric distribution with probability p .
- `beta_cdf(p, n, k)` returns $\text{Prob}(n \leq X \leq k)$.

It turns out that `geometric_cdf(p, n)` = $1 - (1 - p)^n$.

Examples.

- *Input:*

```
geometric_cdf(0.2,3)
```

Output:

```
0.488
```

- *Input:*

```
geometric_cdf(0.2,3,5)
```

Output:

```
0.31232
```

The inverse distribution function for the geometric distribution:
`geometric_icdf`

The `geometric_icdf` command computes the inverse distribution for the geometric distribution.

- *geometric_icdf* takes two arguments:
 - p , a probability (a number between 0 and 1).
 - h , a real number between 0 and 1.
- `geometric_icdf(a, b, h)` returns the inverse distribution for the geometric distribution with probability p ; namely, the smallest natural number n for which $\text{Prob}(X \leq n) \geq h$.

Example.

Input:

```
geometric_icdf(0.2,0.5)
```

Output:

4

9.4.14 The Cauchy distribution

The probability density function for the Cauchy distribution: `cauchy`
`cauchyd`

The `cauchy` (or `cauchyd`) command computes the probability density function for the Cauchy distribution (sometimes called the Lorentz distribution).

- `cauchy` takes two optional arguments and one mandatory argument:
 - Optionally, a and b , real numbers (the parameters; by default $a = 0$ and $b = 1$).
 - x , a real number.
- `cauchy($[a, b,]x$)` returns the value of the density function at x ; namely, $\text{cauchy}(a, b, x) = b/(\pi((x - a)^2 + b^2))$.

Examples.

- *Input:*

```
cauchy(2.2,1.5,0.8)
```

Output:

0.113412073462

- *Input:*

```
cauchy(0.3)
```

Output:

0.292027418517

The cumulative distribution function for the Cauchy distribution:**cauchy_cdf** **cauchydcdf**

The `cauchy_cdf` (or `cauchydcdf`) command computes the cumulative distribution function for the Cauchy distribution.

- *cauchy_cdf* (or *cauchydcdf*) takes three optional arguments and one mandatory argument:
 - Optionally, a and b , the parameters (by default, $a = 0$ and $b = 1$).
 - x , a real number.
 - Optionally, y , a real number.
- `cauchy_cdf([a, b], x)` returns $\text{Prob}(X \leq x)$ for the Cauchy distribution with parameters a and b .
- `cauchy_cdf([a, b], x, y)` returns $\text{Prob}(x \leq X \leq y)$.

It turns out that $\text{cauchy_cdf}(a, b, x) = 1/2 + \arctan((x - a)/b)/\pi$.

Examples.

- *Input:*

```
cauchy_cdf(2, 3, 1.4)
```

Output:

```
0.437167041811
```

- *Input:*

```
cauchy_cdf(1.4)
```

Output:

```
0.802568456711
```

- *Input:*

```
cauchy_cdf(2, 3, -1.9, 1.4)
```

Output:

```
0.228452641651
```

The inverse distribution function for the Cauchy distribution: `cauchy_icdf` `cauchydcidf`

The `cauchy_icdf` (or `cauchydcidf`) command computes the inverse distribution for the Cauchy distribution.

- *cauchy_icdf* (or *cauchydcidf*) takes two optional arguments and one mandatory argument:
 - Optionally, a and b , parameters (by default, $a = 0$ and $b = 1$).

- h , a real number between 0 and 1.
- `cauchy_icdf([a,b], h)` returns the inverse distribution for the Cauchy distribution with parameters a and b ; namely, the value of x for which $\text{Prob}(X \leq x) = h$.

Example.

```
cauchy_icdf(2,3,0.23)
```

Output:

```
-1.40283204777
```

9.4.15 The exponential distribution

The probability density function for the exponential distribution:
`exponential exponentiald`

The exponential distribution depends on one parameters, $\lambda > 0$; the value of the density function at $x \geq 0$ is `exponential(λ, x)` = $\lambda e^{-\lambda x}$. The `exponential` command computes the exponential distribution. `exponentiald` is a synonym for `exponential`.

- `exponential` takes two arguments:
 - λ , a positive number (the parameter).
 - x , a positive number.
- `exponential(λ, x)` returns the value of the exponential density function with parameter λ at x ; namely, `exponential(λ, x)` = $\lambda e^{-\lambda x}$.

Example.

```
exponential(2.1,3.5)
```

Output:

```
0.00134944395675
```

The cumulative distribution function for the exponential distribution: `exponential_cdf exponentiald_cdf`

The `exponential_cdf` (or `exponentiald_cdf`) command computes the cumulative distribution function for the exponential distribution.

- `exponential_cdf` (or `exponentiald_cdf`) takes two arguments:
 - λ , a positive number (the parameter).
 - x , a positive number.
- `exponential_cdf(λ, x)` returns $\text{Prob}(X \leq x)$ for the exponential distribution with parameter λ .

- `exponential_cdf(λ, x, y)` returns $\text{Prob}(x \leq X \leq y)$.

Examples.

- *Input:*

```
exponential_cdf(2.3, 3.2)
```

Output:

```
0.99936380154
```

- *Input:*

```
exponential_cdf(2.3, 0.9, 3.2)
```

Output:

```
0.125549583246
```

The inverse distribution function for the exponential distribution:

`exponential_icdf` `exponentiald_icdf`

The `exponential_icdf` (or `exponentiald_icdf`) command computes the inverse distribution for the exponential distribution.

- *exponential_icdf* (or `exponentiald_icdf`) takes two arguments:
 - λ , a positive number (the parameter).
 - h , a positive real number.
- `exponential_icdf(λ, h)` returns the inverse distribution for the exponential distribution with parameter λ ; namely, the value of x for which $\text{Prob}(X \leq x) = h$.

Example.

Input:

```
exponential_icdf(2.3, 0.87)
```

Output:

```
0.887052534142
```

9.4.16 The Weibull distribution

The probability density function for the Weibull distribution: `weibull`
`weibulld`

The Weibull distribution depends on three parameters; $k > 0$, $\lambda > 0$ and a real number θ . The probability density at x is given by

$$\frac{k}{\lambda} \left(\frac{x - \theta}{\lambda} \right)^{k-1} e^{-((x-\theta)/\lambda)^k} \quad (9.11)$$

The `weibull` (or `weibulld`) command compute this density function.
`weibulld` is a synonym for `weibull`.

- `weibull` takes three mandatory and one optional argument:
 - k , a positive integer.
 - λ , a positive real number.
 - Optionally θ , a real number (by default 0).
 - x , a real number.
- `weibull($k, \lambda[\theta], x$)` returns the value of the Weibull density function, given in (9.11).

Example.*Input:*`weibull(2,1,3)`*or:*`weibull(2,1,0,3)`*Output:*

$$\frac{6}{e^9}$$

The cumulative distribution function for the Weibull distribution:`weibull_cdf weibulld_cdf`

The `weibull_cdf` (or `weibulld_cdf`) command computes the cumulative distribution function for the Weibull distribution.

- `weibull_cdf` (or `weibulld_cdf`) takes three mandatory arguments and two optional arguments:
 - k , a positive integer.
 - λ , a positive real number.
 - Optionally θ , a real number (by default 0).
 - x , a real number.
 - Optionally, y , a real number. If this optional argument is included, then θ must also be included.
- `weibull_cdf($[k,]\lambda[\theta], x$)` returns $\text{Prob}(X \leq x)$ for the Weibull distribution with parameters k, λ and θ .
- `weibull_cdf($[k,]\lambda, \theta, x, y$)` returns $\text{Prob}(x \leq X \leq y)$.

In this case, the Weibull cumulative distribution function is given by the formula $\text{weibull_cdf}(k, \lambda, \theta, x) = 1 - e^{-((x-\theta)/\lambda)^2}$.

Examples.

- *Input:*

`weibull_cdf(2,3,5)`

or:

```
weibull_cdf(2,3,0,5)
```

Output:

$$1 - e^{-\frac{25}{9}}$$

- *Input:*

```
weibull_cdf(2.2,1.5,0.4,1.9)
```

Output:

0.632120558829

- *Input:*

```
weibull_cdf(2.2,1.5,0.4,1.2,1.9)
```

Output:

0.410267239944

The inverse distribution function for the Weibull distribution: `weibull_icdf`
`weibulld_icdf`

The `weibull_icdf` (or `weibulld_icdf`) command computes the inverse distribution for the Weibull distribution.

- `weibull_icdf` (or `weibulld_icdf`) takes three mandatory arguments and one optional argument:
 - k , a positive integer.
 - λ , a positive real number.
 - Optionally θ , a real number (by default 0).
 - h , a real number between 0 and 1.
- `weibull_icdf($k, \lambda[, \theta], h$)` returns the inverse distribution for the Weibull distribution with parameters k, λ and θ ; namely, the value of x for which $\text{Prob}(X \leq x) = h$.

Example.

Input:

```
weibull_icdf(2.2,1.5,0.4,0.632)
```

Output:

1.89977657604

9.4.17 The Kolmogorov-Smirnov distribution: `kolmogorovd`

The density function for the Kolmogorov-Smirnov distribution is given by

$$\text{kolmogorovd}(x) = 1 - 2 \sum_{k=1}^{\infty} (-1)^{k-1} e^{-k^2 x^2} \quad (9.12)$$

The `kolmogorovd` command computes this density function.

- `kolmogorovd` takes one arguments:
 x , a real number.
- `kolmogorovd(x)` returns the density function of the Kolmogorov-Smirnov distribution at x , given by (9.12).

Example.

Input:

```
kolmogorovd(1.36)
```

Output:

```
0.950514123245
```

9.4.18 The Wilcoxon or Mann-Whitney distribution

The Wilcoxon test polynomial: `wilcoxonp`

The `wilcoxonp` command computes the polynomial for the Wilcoxon or Mann-Whitney test.

- `wilcoxonp` takes one mandatory argument and one optional argument:
 - n , an integer.
 - Optionally, k an integer.
- `wilcoxonp(n[,k])` returns the polynomial for the Wilcoxon test.

Examples.

- *Input:*

```
wilcoxonp(4)
```

Output:

```
⌈ 1 1 1 1 1 1 1 1 1 1 1 ⌋
⌊ 16 16 16 8 8 8 8 8 16 16 16 ⌋
```

- *Input:*

```
wilcoxonp(4,3)
```

Output:

```
⌈ 1 1 2 3 4 4 1 4 4 3 2 1 ⌋
⌊ 35 35 35 35 35 35 7 35 35 35 35 35 ⌋
```

The Wilcoxon/Mann-Whitney statistic: wilcoxons

The `wilcoxons` command computes the Wilcoxon or Mann-Whitney statistic.

- `wilcoxons` takes two arguments:
 - L , a list.
 - M , either a list or a real number (a median).
- `wilcoxons(L, M)` returns the Wilcoxon statistic.

Examples.

- *Input:*

```
wilcoxons([1,3,4,5,7,8,8,12,15,17],10)
```

Output:

18

•

```
wilcoxons([1,3,4,5,7,8,8,12,15,17],[2,6,10,11,13,14,15,18,19,20])
```

Output:

128.5

The Wilcoxon or Mann-Whitney test: wilcoxont

The `wilcoxont` command will perform the Wilcoxon or Mann-Whitney test.

- `wilcoxont` takes two mandatory arguments and two optional arguments:
 - L , a sample (list).
 - M , either another sample or a number (a median).
 - Optionally, f , a function.
 - Optionally, x , a real number.
- `wilcoxont($L, M \langle f, x \rangle$)` returns the results of the Wilcoxon test.

Examples.

- *Input:*

```
wilcoxont([1,2,3,4,5,7,8,8,12,15,17],[2,6,10,11,13,14,15,18,19,20])
```

Output:

```

Mann-Whitney 2-sample test, H0 same Median, H1 <>
ranksum 93.0, shifted ranksum 27.0
u1=83 ,u2=27, u=min(u1,u2)=27
Limit value to reject H0 26
P-value 9055/176358 (0.0513444244094), alpha=0.05 H0 not rejected
1

```

- *Input:*

```
wilcoxont([1,3,4,5,7,8,8,12,15,17],[2,6,10,11,13,14,15,18,19,20],0.3)
```

Output:

```

Mann-Whitney 2-sample test, H0 same Median, H1 <>
ranksum 81.5, shifted ranksum 26.5
u1=73.5 ,u2=26.5, u=min(u1,u2)=26.5
Limit value to reject H0 35
P-value 316/4199 (0.0752560133365), alpha=0.3 H0 rejected
0

```

- *Input:*

```
wilcoxont([1,3,4,5,7,8,8,12,15,17] ,10,'>',0.05)
```

Output:

```

Wilcoxon 1-sample test, H0 Median=10, H1 M<>10
Wilcoxon statistic: 18, p-value: 0.375, confidence level: 0.05
1

```

9.4.19 Moment generating functions for probability distributions: mgf

The `mgf` command finds the moment generating function for a probability distribution (such as normal, binomial, poisson, beta, gamma).

- `mgf` takes one or more mandatory arguments:
 - *distd*, the name of the function that finds the distribution's density function.
 - *parameters*, any parameters that would normally be passed to *distd*.
- `mgf(distd,parameters)` returns an expression for the moment generating function for *distd* with parameters *parameters*.

Examples.

- Find the moment generating function for the standard normal distribution.

Input:

`mgf(normald,1,0)`

Output:

e^t

- *Input:*

`mgf(binomial,n,p)`

Output:

$(1 - p + pe^t)^n$

9.4.20 Cumulative distribution functions: `cdf`

The `cdf` command finds the cumulative distribution function for a probability distribution.

- `cdf` takes one or more mandatory arguments and one optional argument.
 - *distd*, the name of the function that finds the distribution's density function.
 - *parameters*, any parameters that would normally be passed to *distd*.
 - *x*, a number.
- `cdf(distd,parameters)` returns an expression for the cumulative distribution function for *distd* with parameters *parameters*.
- `cdf(distd,parameters,x)` returns the value of the cumulative distribution function at *x*. *parameters*.

Examples.

- *Input:*

`cdf(normald,0,1)`

Output:

$\frac{\operatorname{erf}\left(\frac{1}{2}x\sqrt{2}\right) + 1}{2}$

- *Input:*

`cdf(binomial,10,0.5,4)`

Output:

0.376953125

9.4.21 Inverse distribution functions: `icdf`

The `icdf` command finds the inverse cumulative distribution function for a probability distribution.

- `cdf` takes one or more mandatory arguments and one optional argument.
 - *distd*, the name of the function that finds the distribution's density function.
 - *parameters*, any parameters that would normally be passed to *distd*.
 - *x*, a number.
- `icdf(distd,parameters)` returns an expression for the inverse cumulative distribution function for *distd* with parameters *parameters*.
- `icdf(distd,parameters,x)` returns the value of the inverse cumulative distribution function at *x*. *parameters*.

Example.

Input:

```
icdf(normald,0,0.5,0.975)
```

Output:

```
0.97998199227
```

9.4.22 Kernel density estimation: `kernel_density` `kde`

The `kernel_density` command performs kernel density estimation (KDE)¹ `kernel_density` takes a sample, optionally restricted to an interval $[a, b]$, and obtains an estimate \hat{f} of the (unknown) probability density function f from which the samples are drawn. The function \hat{f} is defined by:

$$\hat{f}(x) = \frac{1}{nh} \sum_{i=1}^n K\left(\frac{x - X_i}{h}\right), \quad (9.13)$$

where K is the Gaussian kernel

$$K(u) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}u^2\right)$$

and h is the positive real parameter called the *bandwidth*.

`kde` is a synonym for `kernel_density`.

- `kernel_density` takes one mandatory argument and an unspecified number of optional arguments:

¹For the details on kernel density estimation and its implementation see: Artur Gramacki, *Nonparametric Kernel Density Estimation and Its Computational Aspects*, Springer, 2018.

- L , a list of samples $L = [X_1, X_2, \dots, X_n]$.
- Optionally, a sequence of options from:
 - * **output=type** (or **Output=type** to specify the form of the return value \hat{f} , where *type* can be one of:
 - **exact**, to return \hat{f} as the sum of Gaussian kernels, i.e. as the right side of (9.13), which is usable only when the number of samples is relatively small (up to few hundreds).
 - **piecewise**, to return \hat{f} as a piecewise expression obtained by the spline interpolation of the specified degree (by default, the interpolation is linear) on the interval $[a, b]$ segmented to the specified number of bins.
 - **list** (the default), to return \hat{f} in discrete form, as a list of values $\hat{f}\left(a + k \frac{b-a}{M-1}\right)$ for $k = 0, 1, \dots, M$, where M is the number of bins.
 - * **bandwidth=value**, to specify the bandwidth. *value* can be one of:
 - h , a positive real number.
 - **select** (the default), to have the bandwidth selected using a direct plug-in method,
 - **gauss** (or **normal** or **normald**) to use Silverman's rule of thumb for selecting bandwidth (this method is fast but the results are close to optimal ones only when f is approximately normal).
 - * **bins=n** for a positive integer n (by default 100), the number of bins for simplifying the input data. Only the number of samples in each bin is stored. Bins represent the elements of an equidistant segmentation of the interval S on which KDE is performed. This allows evaluating kernel summations using convolution when **output** is set to **piecewise** or **list**, which significantly lowers the computational burden for large values of n (say, few hundreds or more). If **output** is set to **exact**, this option is ignored.
 - * **a..b** or **range=[a,b]** or **x=a..b** for real numbers a and b , to specify the interval $[a, b]$ on which KDE is performed. If an identifier x is specified, it is used as the variable of the output. If the range endpoints are not specified, they are set to $a = \min_{1 \leq i \leq n} X_i - 3h$ and $b = \max_{1 \leq i \leq n} X_i + 3h$ (unless **output** is set to **exact**, in which case this option is ignored).
 - * **interp=n** for an integer n (by default 1), which specifies the degree of the spline interpolation, ignored unless **output** is set to **piecewise**.
 - * **spline=n** for an integer n , which sets **option** to **piecewise** and **interp** to n .
 - * **eval=x₀** to only return the value $\hat{f}(x_0)$ (this cannot be used with **output** set to **list**).

- * x , an unassigned identifier (by default x) to use as the variable of the output.
- * `exact`, the same as `output=exact`.
- * `piecewise`, the same as `output=piecewise`.

- `kernel_density(L[,options/])` returns the function \hat{f} given in (9.13).

Examples.

- *Input:*

```
kernel_density([1,2,3,2],bandwidth=1/4,exact)
```

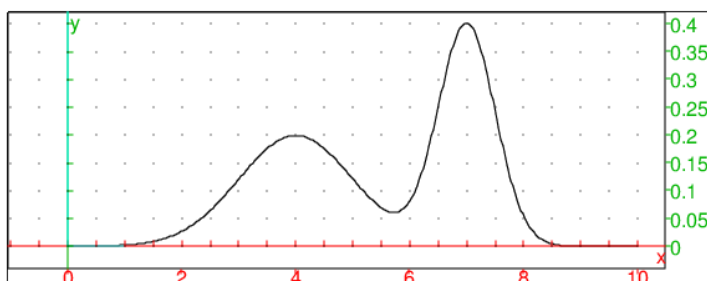
Output:

$$\frac{e^{-\frac{(x-1.0)^2}{0.125}} + e^{-\frac{(x-2.0)^2}{0.125}} + e^{-\frac{(x-3.0)^2}{0.125}} + e^{-\frac{(x-2.0)^2}{0.125}}}{2.50662827463}$$

- *Input:*

```
f:=unapply(normald(4,1,x)/2+normald(7,1/2,x)/2,x);
plot(f(x),x=0..10)
```

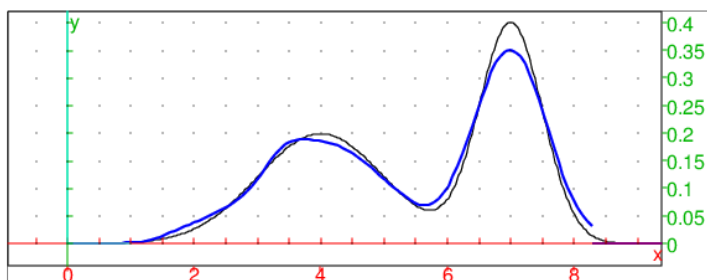
Output:



Input:

```
X:=randvar(f,range=0..10,1000):: S:=sample(X,1000)::
F:=kernel_density(S,piecewise):: plot([F,f(x)],x=0..10,
display=[line_width_2+blue,line_width_1+black])
```

Output:



- *Input:*

```
kernel_density(S,bins=50,spline=3,eval=4.75)
```

Output:

```
0.14655478136
```

- *Input:*

```
time(kernel_density(sample(X,1e5),piecewise))
```

Output:

```
[0.17, 0.1653323]
```

- *Input:*

```
S:=sample(X,5000):;
sqrt(int((f(x)-kde(S,piecewise))^2,x=0..10))
```

Output:

```
0.0269841239243
```

- *Input:*

```
S:=sample(X,25000):;
sqrt(int((f(x)-kde(S,bins=150,piecewise))^2,x=0..10))
```

Output:

```
0.0144212781377
```

9.4.23 Distribution fitting by maximum likelihood: `fitdistr`

The `fitdistr` command finds the parameters for a distribution of a specified type that best fit a set of samples.

- `fitdistr` takes two arguments:
 - L , a list of presumably independent and identically distributed samples.
 - *distr*, a distribution type, which can be one of:
 - * *normal* or *normald*, for a normal distribution.
 - * *exp*, *exponential* or *exponentiald*, for an exponential distribution.
 - * *poisson*, for a Poisson distribution.
 - * *geometric*, for a geometric distribution.
 - * *gammad*, for a gamma distribution.
 - * *betad*, for a beta distribution.
 - * *cauchy* or *cauchy*d, for a Cauchy distribution.

* `weibull` or `weibulld` for a Weibull distribution.

- `fitdistr(L,distr)` returns the name of the specified type of distribution with parameters that fit L most closely according to the method of maximum likelihood.

Examples.

- *Input:*

```
fitdistr(randvector(1000,weibulld,1/2,1),weibull)
```

Output:

```
weibulld(0.517079036032,1.05683817484)
```

- *Input:*

```
X:=randvar(normal,stddev=9.5);;
Y:=randvar(normal,stddev=1.5);;
S:=sample(eval(X/Y,0),1000);; Z:=fitdistr(S,cauchy)
```

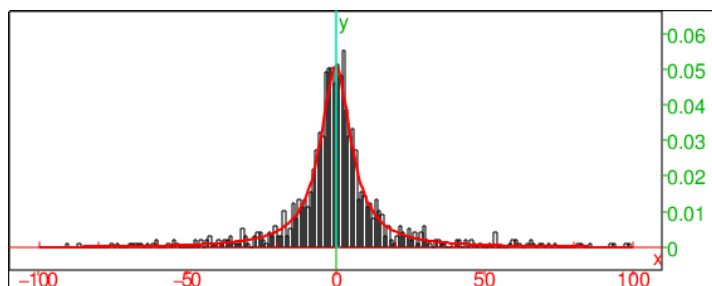
Output:

```
cauchy(0.347058460176,6.55905486387)
```

- *Input:*

```
histogram(select(x->(x>-100 and x<100),S));
plot(Z(x),x=-100..100,display=red+line_width_2)
```

Output:



- *Input:*

```
kolmogorovt(S,Z)
```

Output:

```
["D=",0.0161467485236,"K=",0.510605021406,"1-kolmogorovd(K)=",0.956753826255]
```

The Kolmogorov-Smirnov test indicates that the samples from S are drawn from Z with high probability.

You can fit a lognormal distribution to samples x_1, x_2, \dots, x_n by fitting a normal distribution to the sample logarithms $\log x_1, \log x_2, \dots, \log x_n$ because log-likelihood functions are the same. For example, generate some samples according to the lognormal rule with parameters $\mu = 5$ and $\sigma^2 = 2$:
Input:

```
X:=randvar(normal,mean=5,variance=2);;
S:=sample(eval(exp(X),0),1000);;
```

Then fit the normal distribution to $\log S$:

Input:

```
Y:=fitdistr(log(S),normal)
```

Output:

```
normald(5.04754808715,1.42751619912)
```

The mean of Y is about 5.05 and the variance is about 2.04. Now the variable $Z = \exp(Y)$ has the sought lognormal distribution.

9.4.24 Markov chains: markov

The `markov` command finds characteristic features of a Markov chain.

- `markov` takes one argument:
 M , a transition matrix for a Markov process.
- `markov(M)` returns a sequence consisting of
 - the list of the positive recurrent states.
 - the list of corresponding invariant probabilities.
 - the list of other strong connected components.
 - the list of probabilities of ending up in the sequence of recurrent states.

Example.

Input:

```
markov([[0,0,1/2,0,1/2],[0,0,1,0,0],[1/4,1/4,0,1/4,1/4],[0,0,1/2,0,1/2],[0,0,0,0,1]])
```

Output:

```
[ 4 ], [ 0 0 0 0 1 ], [ 3 1 2 0 ], [ 1
1
1
1
1 ]
```

9.4.25 Generating a random walks: randmarkov

The `randmarkov` command generates random walks or creates stochastic matrices.

To generate a random walk:

- `randmarkov` takes two arguments:
 - M , a transition matrix for a Markov chain.
 - i_0 , an initial state.
 - n , a positive integer.
- `randmarkov(M, i_0, n)` returns a a random walk (given as a vector) starting at i_0 and taking n random steps, where each step is a transition with probabilities given by M .

Example.

Input:

```
randmarkov([[0,1/2,0,1/2],[0,1,0,0],[1/4,1/4,1/4,1/4],[0,0,1/2,1/2]],2,10)
```

Output (for example):

[2, 3, 2, 0, 3, 2, 2, 0, 3, 2, 0]

To create a stochastic matrix:

- `randmarkov` takes two arguments:
 - v , a vector of length p .
 - i_0 , the number of transient states.
- `randmatrix(v, i_0)` returns a stochastic matrix with p recurrent loops (given by v) and i_0 transient states.

Example.

Input:

```
randmarkov([1,2],2)
```

Output (for example):

1.0	0.0	0.0	0.0	0.0
0.0	0.289031975209	0.710968024791	0.0	0.0
0.0	0.46230383289	0.53769616711	0.0	0.0
0.259262238137	0.149948861946	0.143448150524	0.242132758802	0.205207990592
0.231568633749	0.145429586345	0.155664673778	0.282556511895	0.184780594232

9.5 Hypothesis testing

9.5.1 General

Given a random variable X , you may want to know whether some effective parameter p is the same as some expected value p_0 . You will then want to test the hypothesis $p = p_0$, which will be the null hypothesis H_0 . The alternative hypothesis will be H_1 . The tests are:

Two-tailed test This test will reject the hypothesis H_0 if the relevant statistic is outside of a determined interval. This can be denoted ' \neq '.

Left-tailed test This test will reject the hypothesis H_0 if the relevant statistic is less than a specific value. This can be denoted ' $<$ '.

Right-tailed test This test will reject the hypothesis H_0 if the relevant statistic is greater than a specific value. This can be denoted ' $>$ '.

9.5.2 Testing the mean with the Z test: `normalt`

The `normalt` command uses the Z test to test the mean of data.

- `normalt` takes three mandatory arguments and one optional argument:
 - L , a list, which can be one of:
 - * $L = [n_s, n_e]$ for the sample data information, where n_s is the number of successes and n_e is the number of trials n_e .
 - * $L = [m, t]$, where m is the mean and t is the sample size.
 - * L , a data list from a control sample.
 - σ , the standard deviation of the population. If the data list from a control sample is provided, then this argument is unnecessary.
 - *test*, the type of test, one of `!=`, `<` or `>`.
 - Optionally, c , the confidence level (by default 0.05).
- `normalt(L, σ , test, $\langle c \rangle$)` returns the result of a Z test. It will return 0 if the test fails, 1 if the test succeeds, and it will display a summary of the test.

Examples.

- *Input:*

```
normalt([10,30], 0.5, 0.02, '!=', 0.1)
```

Output:

```

*** TEST RESULT 0 ***
Summary Z-Test null hypothesis H0 mu1=mu2, alt. hyp. H1 mu1!=mu2.
Test returns 0 if probability to observe data is less than 0.1
(null hyp. mu1=mu2 rejected with less than alpha probability error)
Test returns 1 otherwise (can not reject null hypothesis)
Data mean mu1=10, population mean mu2=0.5
alpha level 0.1, multiplier*stddev/sqrt(sample size)= 1.64485*0.02/5.47723
0

```

- *Input:*

```
normalt([0.48,50],0.5,0.1,'<')
```

Output:

```

*** TEST RESULT 1 ***
Summary Z-Test null hypothesis H0 mu1=mu2, alt. hyp. H1 mu1<mu2.
Test returns 0 if probability to observe data is less than 0.05
(null hyp. mu1=mu2 rejected with less than alpha probability error)
Test returns 1 otherwise (can not reject null hypothesis)
Data mean mu1=0.48, population mean mu2=0.5
alpha level 0.05, multiplier*stddev/sqrt(sample size)= 1.64485*0.1/7.07107
1

```

9.5.3 Testing the mean with the T test: studentt

The **studentt** command examines whether data conforms to Student's distribution. For small sample sizes, the **studentt** test is preferable to **normalt**.

- **studentt** command takes four mandatory arguments and one optional argument:
 - L , a list, which can be one of:
 - * $L = [n_s, n_e]$ for the sample data information, where n_s is the number of successes and n_e is the number of trials n_e .
 - * $L = [m, t]$, where m is the mean and t is the sample size.
 - * L , a data list from a control sample.
 - μ , the mean of the population to or a data list from a control sample.
 - σ , the standard deviation of the population. If the data list from a control sample is provided, then this argument is unnecessary.
 - *test*, the type of test, one of !=, < or >.
 - Optionally, c , the confidence level (by default 0.05).
- **studentt**($L, \sigma, test \langle, c \rangle$) returns the result of a T test. It will return 0 if the test fails, 1 if the test succeeds, and it will display a summary of the test.

Examples:

- *Input:*

```
studentt([10,20], 0.5, 0.02, '!=', 0.1)
```

Output:

```
*** TEST RESULT 0 ***
Summary T-Test null hypothesis H0 mu1=mu2, alt. hyp. H1 mu1!=mu2.
Test returns 0 if probability to observe data is less than 0.1
(null hyp. mu1=mu2 rejected with less than alpha probability error)
Test returns 1 otherwise (can not reject null hypothesis)
Data mean mu1=10, population mean mu2=0.5, degrees of freedom 20
alpha level 0.1, multiplier*stddev/sqrt(sample size)= 1.32534*0.02/4.47214
0
```

- *Input:*

```
studentt([0.48,20],0.5,0.1,'<')
```

Output:

```
*** TEST RESULT 1 ***
Summary T-Test null hypothesis H0 mu1=mu2, alt. hyp. H1 mu1<mu2.
Test returns 0 if probability to observe data is less than 0.05
(null hyp. mu1=mu2 rejected with less than alpha probability error)
Test returns 1 otherwise (can not reject null hypothesis)
Data mean mu1=0.48, population mean mu2=0.5, degrees of freedom 20
alpha level 0.05, multiplier*stddev/sqrt(sample size)= 1.72472*0.1/4.47214
1
```

9.5.4 Testing a distribution with the χ^2 distribution: `chisquaret`

The `chisquaret` command will use the χ^2 test to compare sample data to a specified distribution.

- `chisquaret` takes one mandatory argument and an unspecified number of optional arguments:
 - L , a list of sample data.
 - Optionally, *distr*. This can be one of
 - * The name of a distribution (see Section 9.3.14 p.747 for a list of distributions and their parameters).
 - * Another list of sample data.

By default this will be the `uniform` distribution.

- *params*, the parameters of the distribution *distr* or the keyword *classes* and optionally *c_{min}* and *c_{dim}*, the minimum size and default size of a statistics class (by default, *class_min* and *class_size*, which themselves default to 0 and 1; see Section 3.5.8 p.76).
- `chisquaret(L⟨,distr⟩)` returns the result of the χ^2 test between the sample data and the named distribution or the two sample data.

Examples.

- *Input:*

```
chisquaret([57,54])
```

Output:

```
Guessing data is the list of number of elements in each class,
adequation to uniform distribution
Sample adequation to a finite discrete probability distribution
Chi2 test result 0.0810810810811,
reject adequation if superior to chisquare_icdf(1,0.95)=3.84145882069 or ch
0.0810810810811
```

- *Input:*

```
chisquaret([1,1,1,1,1,0,0,1,0,1,1],[.4,.6])
```

Output:

```
Sample adequation to a finite discrete probability distribution
Chi2 test result 0.742424242424,
reject adequation if superior to chisquare_icdf(1,0.95)=3.84145882069
or chisquare_icdf(1,1-alpha) if alpha!=5%
0.742424242424
```

- *Input:*

```
chisquaret(ranv(1000,binomial,10,.5),binomial)
```

Output:

```
Binomial: estimating n and p from data 10 0.5055
Sample adequation to binomial(10,0.5055,.), Chi2 test result 7.77825189838,
reject adequation if superior to chisquare_icdf(7,0.95)=14.0671404493
or chisquare_icdf(7,1-alpha) if alpha!=5%
7.77825189838
```

- *Input:*

```
chisquaret(ranv(1000,binomial,10,.5),binomial,11,.5)
```

Output:

```
Sample adequation to binomial(11,0.5,.), Chi2 test result 125.617374161,
reject adequation if superior to chisquare_icdf(10,0.95)=18.3070380533
or chisquare_icdf(10,1-alpha) if alpha!=5%
125.617374161
```

- As an example using `class_min` and `class_size`:

Input:

```
L:= ranv(1000,normald,0,.2)
chisquaret(L,normald,classes,-2,.25)
```

or (setting `class_min` to `-2` and `class_size` to `-0.25` in the graphical configuration):

```
chisquaret(L,normald,classes)
```

Output:

```
Normal density,
estimating mean and stddev from data -0.00345919752912 0.201708100832
Sample adequation to normald_cdf(-0.00345919752912,0.201708100832,.),
Chi2 test result 2.11405080381,
reject adequation if superior to chisquare_icdf(4,0.95)=9.48772903678
or chisquare_icdf(4,1-alpha) if alpha!=5%
2.11405080381
```

In this last case, you are given the value of d^2 of the statistic $D^2 = \sum_{j=1}^k (n_j - e_j)/e_j$, where k is the number of sample classes for `classes(L, -2, 0.25)` (or `classes(L)`), n_j is the size of the j th class, and $e_j = np_j$ where n is the size of `L` and p_j is the probability of the j th class interval assuming a normal distribution with the mean and population standard deviation of `L`.

9.5.5 Testing a distribution with the Kolmogorov-Smirnov distribution: `kolmogorovt`

The `kolmogorovt` command uses the Kolmogorov test to compare sample data to a specified continuous distribution.

- `kolmogorovt` takes two arguments and possibly additional parameters.
 - `L`, a list of sample data.
 - Optionally, *distr*. This can be one of

- * The name of a distribution and the necessary parameters (see Section 9.3.14 p.747 for a list of distributions and their parameters).
 - * Another list of sample data.
- `kolmogorovt(L, distr)` returns a list of three values:
 - The D statistic, which is the maximum distance between the cumulative distribution functions of the samples or the sample and the given distribution.
 - The K value, where $K = D\sqrt{n}$ (for a single data set, where n is the size of the data set) or $K = D\sqrt{n_1n_2/(n_1 + n_2)}$ (when there are two data sets, with sizes n_1 and n_2). The K value will tend towards the Kolmogorov-Smirnov distribution as the size of the data set goes to infinity.
 - `1 - kolmogorovd(K)`, which will be close to 1 when the distributions look like they match.

Examples.

- *Input:*

```
kolmogorovt(randvector(100,normald,0,1),normald(0,1))
```

Output (for example):

```
[D=0.112141597243, K=1.12141597243, 1-kolmogorovd(K)=, 0.161616499536]
```

- *Input:*

```
kolmogorovt(randvector(100,normald,0,1),student(2))
```

Output (for example):

```
[D=0.112592987625, K=1.12592987625, 1-kolmogorovd(K)=0.158375510292]
```

Chapter 10

Numerical computations

Real numbers may have an exact representation (e.g. rationals, symbolic expressions involving square roots or constants like π , ...) or approximate representation, which means that internally the real is represented by a rational (with a denominator that is a power of the basis of the representation) close to the real. Inside **Xcas**, the standard scientific notation is used for approximate representation; that is a mantissa (with a point as decimal separator) optionally followed by the letter **e** and an integer exponent.

Note that the real number 10^{-4} is an exact number but $1e-4$ is an approximate representation of this number.

10.1 Floating point representation.

This section discusses how real numbers are represented.

10.1.1 Digits

The **Digits** variable (see Section 3.5.1 p.70) is used to control how real numbers are represented and also how they are displayed. When the specified number of digits is less or equal to 14 (for example **Digits:=14**), then hardware floating point numbers are used and they are displayed using the specified number of digits. When **Digits** is larger than 14, **Xcas** uses the MPFR library, the representation is similar to hardware floats (cf. infra) but the number of bits of the mantissa is not fixed and the range of exponents is much larger. More precisely, the number of bits of the mantissa of a created MPFR float is $\text{ceil}(\text{Digits} \cdot \log(10) / \log(2))$.

Note that if you change the value of **Digits**, this will affect the creation of new real numbers compiled from command lines or programs or by instructions like **approx**, but it will not affect existing real numbers. Hence hardware floats may coexist with MPFR floats, and even in MPFR floats, some may have 100 bits of mantissa and some may have 150 bits of mantissa. If operations mix different kinds of floats, the most precise kind of floats are coerced to the less precise kind of floats.

10.1.2 Representation by hardware floats

A real is represented by a floating number d , that is

$$d = 2^\alpha * (1 + m), \quad 0 < m < 1, -2^{10} < \alpha < 2^{10}$$

If $\alpha > 1 - 2^{10}$, then $m \geq 1/2$, and d is a normalized floating point number, otherwise d is denormalized ($\alpha = 1 - 2^{10}$). The special exponent 2^{10} is used to represent plus or minus infinity and NaN (Not a Number). A hardware float is made of 64 bits:

- the first bit is for the sign of d (0 for '+' and 1 for '-')
- the 11 following bits represents the exponent, more precisely if α denotes the integer given by the 11 bits, the exponent is $\alpha + 2^{10} - 1$,
- the 52 last bits codes the mantissa m , more precisely if M denotes the integer given by the 52 bits, then $m = 1/2 + M/2^{53}$ for normalized floats and $m = M/2^{53}$ for denormalized floats.

Examples of representations of the exponent:

- $\alpha = 0$ is coded by 011 1111 1111
- $\alpha = 1$ is coded by 100 0000 0000
- $\alpha = 4$ is coded by 100 0000 0011
- $\alpha = 5$ is coded by 100 0000 0100
- $\alpha = -1$ is coded by 011 1111 1110
- $\alpha = -4$ is coded by 011 1111 1011
- $\alpha = -5$ is coded by 011 1111 1010
- $\alpha = 2^{10}$ is coded by 111 1111 1111
- $\alpha = 2^{-10} - 1$ is coded by 000 0000 000

Remark: $2^{-52} = 0.2220446049250313e - 15$

10.1.3 Examples of representations of normalized floats

- 3.1:

We have:

$$\begin{aligned} 3.1 &= 2 * (1 + \frac{1}{2} + \frac{1}{2^5} + \frac{1}{2^6} + \frac{1}{2^9} + \frac{1}{2^{10}} + \dots) \\ &= 2 * (1 + \frac{1}{2} + \sum_{k=1}^{\infty} (\frac{1}{2^{4*k+1}} + \frac{1}{2^{4*k+2}})) \end{aligned}$$

hence $\alpha = 1$ and $m = \frac{1}{2} + \sum_{k=1}^{\infty} (\frac{1}{2^{4*k+1}} + \frac{1}{2^{4*k+2}})$. Hence the hexadecimal and binary representation of 3.1 is:

40 (01000000), 8 (00001000), cc (11001100), cc (11001100),
cc (11001100), cc (11001100), cc (11001100), cd (11001101),

the last octet is 1101, the last bit is 1, because the following digit is 1 (upper rounding).

- 3.:

We have $3 = 2 * (1 + 1/2)$. Hence the hexadecimal and binary representation of 3 is:

40 (01000000), 8 (00001000), 0 (00000000), 0 (00000000),
0 (00000000), 0 (00000000), 0 (00000000), 0 (00000000)

10.1.4 Difference between the representation of (3.1-3) and of 0.1

For the representation of 0.1:

$$\begin{aligned} 0.1 &= 2^{-4} \cdot (1 + \frac{1}{2} + \frac{1}{2^4} + \frac{1}{2^5} + \frac{1}{2^8} + \frac{1}{2^9} + \dots) \\ &= 2^{-4} \cdot \sum_{k=0}^{\infty} (\frac{1}{2^{4*k}} + \frac{1}{2^{4*k+1}}) \end{aligned}$$

hence $\alpha = 1$ and

$$m = \frac{1}{2} + \sum_{k=1}^{\infty} (\frac{1}{2^{4*k}} + \frac{1}{2^{4*k+1}}),$$

therefore the representation of 0.1 is

3f (00111111), b9 (10111001), 99 (10011001), 99 (10011001),
99 (10011001), 99 (10011001), 99 (10011001), 9a (10011010),

the last octet is 1010, indeed the 2 last bits 01 became 10 because the following digit is 1 (upper rounding).

For the representation of $a := 3.1 - 3$:

Computing a is done by adjusting exponents (here nothing to do), then subtracting the mantissa and adjusting the exponent of the result to have a normalized float. The exponent is $\alpha = -4$ (that corresponds at $2 \cdot 2^{-5}$) and the bits corresponding to the mantissa begin at $1/2 = 2 \cdot 2^{-6}$: the bits of the mantissa are shifted to the left 5 positions and you get:

3f (00111111), b9 (10111001), 99 (10011001), 99 (10011001),
99 (10011001), 99 (10011001), 99 (10011001), a0 (10100000),

Therefore, $a > 0.1$ and $a - 0.1 = 1/2^{50} + 1/2^{51}$ (since 100000-11010=110).

Remark:

This is the reason why:

Input:

floor(1/(3.1-3))

returns 9 and not 10 when Digits:=14.

10.2 Approximate evaluation: evalf approx Digits

The `evalf` command `evalf` or `approx`, if possible, evaluates to a numeric approximation (see Section 6.8.1 p.171). The approximation is to `Digits` digits (see Section 3.5.1 p.70), this can be changed with an optional second argument.

Examples.

- *Input:*

```
evalf(sqrt(2))
```

Output: (Assuming that in the `cas` configuration (`Cfg` menu) `Digits=7`, so hardware floats are used, and 7 digits are displayed)

```
1.414214
```

- You can change the number of digits in a command line by assigning the variable `DIGITS` or `Digits`.

Input:

```
DIGITS:=20
evalf(sqrt(2))
```

Output:

```
1.4142135623730950488
```

- *Input:*

```
evalf(10^-5)
```

Output:

```
1e-05
```

- *Input:*

```
evalf(10^15)
```

Output:

```
1e+15
```

- *Input:*

```
evalf(sqrt(2))*10^-5
```

Output:

```
1.41421356237e-05
```


10.3 Numerical algorithms

10.3.1 Approximating solution of an equation: newton

The `newton` command uses Newton's method to approximate a solution to an equation.

- `newton` takes one mandatory argument and four optional arguments:
 - ex , an expression.
 - Optionally, var , the variable used in this expression (by default `x`).
 - Optionally, a , a number (by default 0)
 - Optionally, ϵ , a small number (by default `1e-8`)
 - Optionally, $nbiter$ (by default 12).
- `newton($ex, \langle var, a, \epsilon, nbiter \rangle$)` returns an approximate solution of the equation $ex=0$ using the Newton algorithm with starting point $var=a$. The maximum number of iterations is $nbiter$ and the precision is ϵ .

Examples.

- *Input:*

```
newton(x^2-2,x,1)
```

Output:

```
1.41421356237
```

- *Input:*

```
newton(x^2-2,x,-1)
```

Output:

```
-1.41421356237
```

- *Input:*

```
newton(cos(x)-x,x,0)
```

Output:

```
0.739085133215
```

10.3.2 Approximating computation of the derivative number: `nDeriv`

The `nDeriv` command numerically approximates the value of a derivative.

- `nDeriv` takes as arguments:
 - *expr*, expression.
 - Optionally, *var*, the variable used in the expression (by default `x`).
 - Optionally, *h* (by default 0.001).
- `nDeriv(expr <var, h>)` returns an approximated value of the derivative of the expression *expr* at the point *var* using the formula:

$$\frac{f(var + h) - f(var - h)}{2h}$$

Examples.

- *Input:*

```
nDeriv(x^2,x)
```

Output:

$$\left((x + 0.001)^2 - (x - 0.001)^2 \right) \cdot 500.0$$

- *Input:*

```
subst(nDeriv(x^2,x),x=1)
```

Output:

2.0

- *Input:*

```
nDeriv(exp(x^2),x,0.00001)
```

Output:

$$\left(e^{(x+1.0 \times 10^{-5})^2} - e^{(x-1.0 \times 10^{-5})^2} \right) \cdot 50000.0$$

- *Input:*

```
subst(exp(nDeriv(x^2,x,0.00001)),x=1)
```

Output:

7.38905609706

which is an approximate value of $e^2 \approx 7.38905609893$.

10.3.3 Approximating computation of integrals: `romberg` `nInt`

The `romberg` command finds approximate values of integrals using the Romberg method.

`nInt` is a synonym for `rombert`.

- `romberg` takes three mandatory arguments and one optional argument:
 - *expr*, an expression involving one variable.
 - Optionally, *var*, the variable (by default *x*).
 - *a, b*, two real numbers.
- `romberg(expr⟨,var⟩a,b)` returns an approximated value of the integral $\int_a^b expr dvar$. The integrand must be sufficiently regular for the approximation to be accurate, otherwise, `romberg` returns a list of real values that come from the application of the Romberg algorithm (the first list element is the trapezoid rule approximation, the next ones come from the application of the Euler-MacLaurin formula to remove successive even powers of the step of the trapezoid rule).

Example.

Input:

```
romberg(exp(x^2),x,0,1)
```

Output:

```
1.46265174591
```

10.3.4 Approximating integrals with an adaptive Gaussian quadrature at 15 points: `gaussquad`

The `gaussquad` command finds an approximate value of an integral, calculated by an adaptive method by Ernst Hairer which uses a 15-point Gaussian quadrature.

- `gaussquad` takes four arguments:
 - *expr*, an expression.
 - *var*, the variable used by the expression.
 - *a, b*, two numbers.
- `gaussquad(expr⟨,var⟩a,b)` returns an approximation of the integral $\int_a^b expr dvar$.

Examples.

- *Input:*

```
gaussquad(exp(x^2),x,0,1)
```

Output:

1.46265174591

- *Input:*

`gaussquad(exp(-x^2), x, -1, 1)`

Output:

1.49364826562

10.3.5 Approximating solutions of $y'=f(t,y)$: `odesolve`

The `odesolve` command can solve first order differential equations or first order systems. This section covers equations, systems will be discussed in the next section.

`odesolve` finds values of the solution of a differential equation of the form $y' = f(t, y)$; specifically, it will approximate $y(t_1)$ for a specified t_1 .

`odesolve` can takes its arguments in various ways. Letting t and y be the independent and dependent variables, t_0 and y_0 be the initial values, t_1 the place where you want the value of y , f be the function in the differential equation, $f(t, y)$ be an expression which determines the function f (see Section 6.15.1 p.234 for the difference between a function and an expression):

- `odesolve` takes three or four mandatory arguments and two optional arguments:
 - *mandatory*, mandatory arguments given by one of the following sequences:
 - * $f(t, y), [t, y], [t_0, y_0], t_1$
 - * $f(t, y), t = t_0..t_1, y, y_0$
 - * $t_0..t_1, f, y_0$
 - * $t_0..t_1, (t, y) \rightarrow f(t, y), y_0$
 - Optionally, `tstep=n`, to set the initial `tstep` value to the numeric solver from the GSL (Gnu Scientific Library). It may be modified by the solver. It is also used to control the number of iterations of the solver by $2(t_1 - t_0)/n$ (if the number of iterations exceeds this value, the solver will stops at a time $t < t_1$).
 - Optionally, `curve`, the symbol.
- `odesolve(mandatory <, tstep=n, curve >)` returns an approximate value of $y(t_1)$ where $y(t)$ is the solution of:

$$y'(t) = f(t, y(t)), \quad y(t_0) = y_0$$

With an optional argument of `curve`, the list of all the $[t, [y(t)]]$ values that were computed are returned.

Examples.

- *Input:*

```
odesolve(sin(t*y),[t,y],[0,1],2)
```

or:

```
odesolve(sin(t*y),t=0..2,y,1)
```

or:

```
odesolve(0..2,(t,y)->sin(t*y),1)
```

or:

```
f(t,y):=sin(t*y)
odesolve(0..2,f,1)
```

Output:

```
[1.82241255674]
```

• *Input:*

```
odesolve(0..2,f,1,tstep=0.3)
```

Output:

```
[1.82241255675]
```

• *Input:*

```
odesolve(sin(t*y),t=0..2,y,1,tstep=0.5)
```

Output:

```
[1.82241255674]
```

• *Input:*

```
odesolve(sin(t*y),t=0..2,y,1,tstep=0.5,curve)
```

Output:

```
[
  0.0          [1.0]
  0.0238917513909 [1.00028543504]
  0.065808814858  [1.00216696089]
  0.108895370376  [1.00594077449]
  ⋮              ⋮
  1.96462490594   [1.8389834135]
  1.97769352646   [1.83297839039]
  1.9908403154    [1.82679805346]
  2.0             [1.82241255674]
]
```

10.3.6 Approximating solutions of the system $\mathbf{v}' = \mathbf{f}(t, \mathbf{v})$: `odesolve`

This section covers using `odesolve` to solve first order systems of differential equations; using it to solve a single first order differential equation was discussed last section.

The `odesolve` can be used to solve a system of the form

$$x' = f(t, x)$$

where $x = [x_1, \dots, x_n]$ is a list of unknown functions and f is a function of $n + 1$ variables with an initial condition.

`odesolve` can takes its arguments in various ways. Letting t be the independent variable and $x = [x_1, \dots, x_n]$ be a vector of dependent variables, t_0 and x_0 be the initial values, t_1 the place where you want the value of x , f be the function in the differential equation, $f(t, x)$ be a list of expressions which determines the function f (see Section 6.15.1 p.234 for the difference between a function and an expression):

- `odesolve` takes three or four mandatory arguments and two optional arguments:
 - *mandatory*, mandatory arguments given by one of the following sequences:
 - * $f(t, x), t = t_0..t_1, x, x_0$
 - * $t_0..t_1, (t, y) \rightarrow f(t, y), x_0$
 - * $t_0..t_1, f, x_0$
 - Optionally, `curve`, the symbol.
- `odesolve(mandatory [, curve])` returns an approximate value of $x(t_1)$ where $x(t)$ is the solution of:

$$x'(t) = f(t, x(t)), \quad x(t_0) = x_0$$

With an optional argument of `curve`, the list of all the $[t, [x(t)]]$ values that were computed by the solver are returned.

Examples.

- Solve the system:

$$\begin{aligned} x'(t) &= -y(t) \\ y'(t) &= x(t) \end{aligned}$$

Input:

```
odesolve([-y,x], t=0..pi, [x,y], [0,1])
```

Output:

```
[-1.79045146764 × 10-15, -1]
```

- Solve the system:

$$\begin{aligned}x'(t) &= -y(t) \\ y'(t) &= x(t)\end{aligned}$$

Input:

```
odesolve(0..pi, (t,v)->[-v[1],v[0]], [0,1])
```

or:

```
f(t,v):=[-v[1],v[0]]
odesolve(0..pi,f,[0,1])
```

Output:

$$[-1.79045146764 \times 10^{-15}, -1]$$

- *Input:*

```
odesolve(0..pi/4,f,[0,1],curve)
```

Output:

$$\begin{bmatrix} 0.0 & [0.0, 1.0] \\ 0.0165441856471 & [-0.0165434309391, 0.999863148082] \\ 0.0325491321983 & [-0.0325433851614, 0.999470323763] \\ 0.0486049854945 & [-0.0485858499906, 0.998819010222] \\ \vdots & \vdots \\ 0.747336757246 & [-0.679687679865, 0.733501641333] \\ 0.76509544295 & [-0.692605846268, 0.721316256377] \\ 0.78286231703 & [-0.705311395415, 0.708897619897] \\ 0.785398163397 & [-0.707106781186, 0.707106781186] \end{bmatrix}$$

10.3.7 Approximating solutions of nonlinear second-order boundary value problems: `bvpsolve`

The `bvpsolve` command finds an approximate solution of a boundary value problem

$$y'' = f(x, y, y'), \quad y(a) = \alpha, \quad y(b) = \beta$$

on an interval $[a, b]$. The procedure uses the method of nonlinear shooting which is based on Newton and Runge-Kutta methods. Values of y and its first derivative y' are approximated at points $x_k = a + k\delta$, where $\delta = \frac{b-a}{N}$ and $k = 0, 1, \dots, N$. For the numeric tolerance (precision) threshold, the algorithm uses `epsilon` specified in the session settings in Xcas (see `secrefssec:confcomp`, item 9).

- `bvpsolve` takes three mandatory arguments and four optional arguments:

- $f(x, y, y')$, an expression involving an independent variable x , a dependent variable y and y' .
 - $[x = a..b, y]$, a list specifying the independent variable x , its range $[a, b]$ and the sought function y
 - $[\alpha, \beta]$, a list of the boundary values of y .
 - Optionally, A , an initial guess for $y'(a)$ (by default, $(\beta - \alpha)/(b - a)$).
 - Optionally, N , an integer greater than or equal to 2 (by default 100).
 - `output=type` or `Output=type`, the type of the output, where *type* can be one of:
 - * `list`
 - * `diff`
 - * `piecewise`
 - * `spline`
 (By default `list`).
 - `limit=M`, a positive integer for a limit for the number of iterations before the procedure is stopped (by default there is no limit).
- `bvpsolve(f(x, y, y'), [x = a..b, y], [\alpha, \beta] \langle, A, N, output=type, limit=M \rangle)` returns, for the different output types:
- `list`, a list of pairs $[x_k, y_k]$ where $y_k \approx y(x_k)$,
 - `diff`, a list of lists $[x_k, y_k, y'_k]$, where $y'_k \approx y'(x_k)$,
 - `piecewise`, a piecewise linear interpolation of the points (x_k, y_k) .
 - `spline`, a piecewise spline interpolation of the points (x_k, y_k) , based on the values y'_k computed in the process.

Note that the shooting method is sensitive to roundoff errors and may fail to converge in some cases, especially when y is a rapidly increasing function. In the absence of convergence or if the maximum number of iterations is exceeded, `bvpsolve` returns `undef`. However, if the output type is `list` or `piecewise` and if $N > 2$, a slower but more stable finite-difference method (which approximates only the function y) is tried first.

Sometimes setting an initial guess A for $y'(a)$ to a suitable value may help the shooting algorithm to converge or to converge faster.

Examples.

- Solve the problem

$$y'' = \frac{1}{8} (32 + 2x^3 - y y'), \quad 1 \leq x \leq 3$$

with boundary conditions $y(1) = 17$ and $y(3) = \frac{43}{3}$. Use $N = 20$, which gives an x -step of 0.01.

Input:

k	x_k	y_k	$y(x_k)$
0	1.0	17.0	17.0
1	1.1	15.7554961579	15.7554545455
2	1.2	14.7733911821	14.7733333333
3	1.3	13.9977543159	13.9976923077
4	1.4	13.388631813	13.3885714286
5	1.5	12.9167227424	12.9166666667
6	1.6	12.5600506483	12.56
7	1.7	12.3018096101	12.3017647059
8	1.8	12.1289281414	12.1288888889
9	1.9	12.0310865274	12.0310526316
10	2.0	12.0000289268	12.0
11	2.1	12.0290719981	12.029047619
12	2.2	12.1127475278	12.1127272727
13	2.3	12.2465382803	12.2465217391
14	2.4	12.4266798825	12.4266666667
15	2.5	12.650010254	12.65
16	2.6	12.9138537834	12.9138461538
17	2.7	13.2159312426	13.2159259259
18	2.8	13.5542890043	13.5542857143
19	2.9	13.9272429048	13.9272413793
20	3.0	14.3333333333	14.3333333333

Table 10.1: approximate and true values of the function $y = x^2 + 16/x$ on $[1, 3]$

```
bvpsolve((32+2x^3-y*y')/8,[x=1..3,y],[17,43/3],20)
```

The output is shown in Table 10.1 (the middle two columns) alongside with the values $y(x_k)$ of the exact solution $y = x^2 + 16/x$ (the fourth column).

- Solve

$$y'' = \frac{x^2(y')^2 - 9y^2 + 4x^6}{x^5}, \quad 1 \leq x \leq 2,$$

with the boundary conditions $y(1) = 0$ and $y(2) = \ln 256$. Obtain the solution as a piecewise spline interpolation for $N = 10$ and estimate the absolute error **err** of the approximation using the exact solution $y = x^3 \ln x$ and the **romberg** command for numerical integration. You need to explicitly set an initial guess A for the value $y'(1)$ because the algorithm fails to converge with the default guess $A = \ln 256 \approx 5.545$. Therefore let $A = 1$ instead.

Input:

```
f:=(x^2*diff(y(x),x)^2-9*y(x)^2+4*x^6)/x^5;;vars:=[x=1..2,y];;
yinit:=[0,ln(256),1];;
p:=bvpsolve(f,vars,yinit,10,output=spline);;
err:=sqrt(romberg((p-x^3*ln(x))^2,x=1..2))
```

Output:

$$3.27751904973 \times 10^{-6}$$

Note that, if the output type was set to `list` or `piecewise`, the solution would have been found even without specifying an initial guess for $y'(1)$ because the algorithm would automatically apply the alternative finite-difference method, which converges.

10.4 Solving equations with `fsolve` `nSolve` `cfsolve`

The `fsolve` command can solve equations or systems of equations. This section will discuss solving equations; systems will be discussed in the next section.

The `cfsolve` command is the complex version of `fsolve`, with the same arguments. The only difference is that `cfsolve` gives numeric solutions over the complex numbers, even if `Xcas` is not in complex mode (see Section 3.5.5 p.72). `fsolve` will return complex roots, but only in complex mode.

`fsolve` solves numeric equations of the form:

$$f(x) = 0, \quad x \in (a, b)$$

Unlike `solve` (Section 6.55.6 p.619) or `proot` (Section 10.6 p.824), it is not limited to polynomial equations.

`nSolve` is a synonym for `fsolve`.

- `fsolve` takes one mandatory argument and three optional arguments:
 - *eqn*, an equation involving one variable.
 - Optionally, *var*, the variable (by default `x`).
 - Optionally, *init*, an initial approximation or range.
 - Optionally, *algorithm*, the name of an iterative algorithm to be used by the GSL solver.
- `fsolve(eqn,var,init[,algorithm])` returns an approximate solution to the equation.

Examples.

- *Input (in real mode):*

```
fsolve(x^3-1,x)
```

Output:

```
1.0
```

- *Input (in complex mode):*

```
fsolve(x^3-1,x)
```

Output:

```
[-0.5 - 0.866025403784i, -0.5 + 0.866025403784i, 1.0]
```

- *Input (in any mode):*

```
cfsolve(x^3-1,x)
```

Output:

```
[-0.5 - 0.866025403784i, -0.5 + 0.866025403784i, 1.0]
```

- *Input:*

```
fsolve(sin(x)=2)
```

Output:

```
[]
```

- *Input:*

```
cfsolve(sin(x)=2)
```

Output:

```
[1.57079632679 - 1.31695789692i, 1.57079632679 + 1.31695789692i]
```

The different values of *algorithm* are explained in the rest of this section.

10.4.1 fsolve with the option bisection_solver

This algorithm of dichotomy is the simplest but also generically the slowest. It encloses the zero of a function on an interval. Each iteration cuts the interval into two parts, the middle point value is calculated. The function sign at this point gives you the half-interval on which the next iteration will be performed.

Example.

Input:

```
fsolve((cos(x))=x,x,-1..1,bisection_solver)
```

Output:

```
[0.739085133215]
```

10.4.2 fsolve with the option brent_solver

The Brent method interpolates of f at three points, finds the intersection of the interpolation with the x axis, computes the sign of f at this point and chooses the interval where the sign changes. It is generically faster than bisection.

Example.

Input:

```
fsolve((cos(x))=x,x,-1..1,brent_solver)
```

Output:

```
[0.739085133215]
```

10.4.3 fsolve with the option falsepos_solver

The "false position" algorithm is an iterative algorithm based on linear interpolation: it computes the value of f at the intersection of the line $(a, f(a))$, $(b, f(b))$ with the x axis. This value gives us the part of the interval containing the root, and on which a new iteration is performed. The convergence is linear but generically faster than bisection.

Example.

Input:

```
fsolve((cos(x))=x,x,-1..1,falsepos_solver)
```

Output:

```
[0.739085133215]
```

10.4.4 fsolve with the option newton_solver

newton_solver is the standard Newton method. The algorithm starts at an initial value x_0 , then finds the intersection x_1 of the tangent at x_0 to the graph of f , with the x axis, the next iteration is done with x_1 instead of x_0 . The x_i sequence is defined by

$$x_0 = x_0, \quad x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

If the Newton method converges, it is a quadratic convergence for roots of multiplicity 1.

Example.

Input:

```
fsolve((cos(x))=x,x,0,newton_solver)
```

Output:

```
0.739085133215
```

10.4.5 fsolve with the option secant_solver

The secant method is a simplified version of Newton's method. The computation of x_1 is done using Newton's method, but then the computation of $f'(x_n)$, $n > 1$ is done approximately. This method is used when the computation of the derivative is expensive:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'_{est}}, \quad f'_{est} = \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}}$$

The convergence for roots of multiplicity 1 is of order $(1 + \sqrt{5})/2 \approx 1.62 \dots$

Examples.

• *Input:*

```
fsolve((cos(x))=x,x,-1..1,secant_solver)
```

Output:

[0.739085133215]

- *Input:*

```
fsolve((cos(x))=x,x,0,secant_solver)
```

Output:

0.739085133215

10.4.6 fsolve with the option `steffenson_solver`

The Steffenson method is generically the fastest method. It combines Newton's method with a "delta-two" Aitken acceleration: with Newton's method, you get the sequence x_i and the convergence acceleration gives the Steffenson sequence

$$R_i = x_i - \frac{(x_{i+1} - x_i)^2}{(x_{i+2} - 2x_{i+1} + x_i)}$$

Example.

Input:

```
fsolve(cos(x)=x,x,0,steffenson_solver)
```

Output:

0.739085133215

10.5 Solving systems with `fsolve` and `cfsolve`

The previous section discussed using `fsolve` to solve equations. This section will discuss systems of equations.

As before, the `cfsolve` command is the complex version of `fsolve`, with the same arguments. The only difference is that `cfsolve` gives numeric solutions over the complex numbers, even if `Xcas` is not in complex mode (see Section 3.5.5 p.72). `fsolve` will return complex roots, but only in complex mode.

For solving systems of equations:

- `fsolve` takes three mandatory arguments and one optional argument:

- *eqns*, a list of equations (or expressions, considered to be equal to zero) to solve.
- *vars*, a list of the variables.
- *init*, a list initial values for the variables.
- Optionally, *method*, the method to use. The possible methods are:

```
* dnewton_solver
* hybrid_solver
* hybrids_solver
```

```
* newtonj_solver
* hybridj_solver
* hybridsj_solver
```

- `fsolve(eqns,vars,init[,method])` returns an approximate solution to *eqns*.

Examples.

- *Input (in real mode):*

```
fsolve([x^2+y+1,x+y^2-1],[x,y])
```

Output:

$$\begin{bmatrix} 0.0 & -1.0 \\ -0.453397651516 & -1.2055694304 \end{bmatrix}$$

- *Input (in complex mode):*

```
fsolve([x^2+y+1,x+y^2-1],[x,y])
```

Output:

$$\begin{bmatrix} 0.0 & -1.0 \\ 0.226698825758 - 1.46771150871i & 1.1027847152 + 0.665456951153i \\ 0.226698825758 + 1.46771150871i & 1.1027847152 - 0.665456951153i \\ -0.453397651516 & -1.2055694304 \end{bmatrix}$$

- *Input (in any mode):*

```
cfsolve([x^2+y+1,x+y^2-1],[x,y])
```

Output:

$$\begin{bmatrix} 0.0 & -1.0 \\ 0.226698825758 - 1.46771150871i & 1.1027847152 + 0.665456951153i \\ 0.226698825758 + 1.46771150871i & 1.1027847152 - 0.665456951153i \\ -0.453397651516 & -1.2055694304 \end{bmatrix}$$

- *Input:*

```
cfsolve([x^2+y+2,x+y^2+2],[x,y])
```

Output:

$$\begin{bmatrix} 0.5 + 1.65831239518i & 0.5 - 1.65831239518i \\ 0.5 - 1.65831239518i & 0.5 + 1.65831239518i \\ -0.5 + 1.32287565553i & -0.5 + 1.32287565553i \\ -0.5 - 1.32287565553i & -0.5 - 1.32287565553i \end{bmatrix}$$

The methods are inherited from the GSL. The methods whose names end with `j_solver` use the jacobian matrix, the rest use approximations for the derivatives.

All methods use an iteration of Newton kind

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

The four methods `hybrid*_solver` use also a method of gradient descent when the Newton iteration would make a too large of a step. The length of the step is computed without scaling for `hybrid_solver` and `hybridj_solver` or with scaling (computed from $f'(x_n)$) for `hybrids_solver` and `hybridsj_solver`.

The rest of this section will cover the various `method` options.

10.5.1 fsolve with the option `dnewton_solver`

Example.

Input:

```
fsolve([x^2+y-2,x+y^2-2],[x,y],[2,2],dnewton_solver)
```

Output:

```
[1.0, 1.0]
```

10.5.2 fsolve with the option `hybrid_solver`

Example.

Input:

```
fsolve([x^2+y-2,x+y^2-2],[x,y],[2,2],hybrid_solver)
```

Output:

```
[1.0, 1.0]
```

10.5.3 fsolve with the option `hybrids_solver`

Example.

Input:

```
fsolve([x^2+y-2,x+y^2-2],[x,y],[2,2],hybrids_solver)
```

Output:

```
[1.0, 1.0]
```

10.5.4 fsolve with the option `newtonj_solver`

Example.

Input:

```
fsolve([x^2+y-2,x+y^2-2],[x,y],[0,0],newtonj_solver)
```

Output:

```
[1.0, 1.0]
```

10.5.5 fsolve with the option hybridj_solver**Example.***Input:*

```
fsolve([x^2+y-2,x+y^2-2],[x,y],[2,2],hybridj_solver)
```

Output:

```
[1.0,1.0]
```

10.5.6 fsolve with the option hybridsj_solver**Example.***Input:*

```
fsolve([x^2+y-2,x+y^2-2],[x,y],[2,2],hybridsj_solver)
```

Output:

```
[1.0,1.0]
```

10.6 Numeric roots of a polynomial: proot

The **proot** command numerically finds the roots of a squarefree polynomial.

- **proot** takes one argument:
 P , a squarefree polynomial, either in symbolic form or as a list of polynomial coefficients (written by decreasing order).
- **proot**(P) returns a list of the numeric roots of P .

Examples.

- *Input:* Find the numeric roots of $P(x) = x^3 + 1$:

Input:

```
proot([1,0,0,1])
```

or:

```
proot(x^3+1)
```

Output:

```
[-1.0, 0.5 - 0.866025403784i, 0.5 + 0.866025403784i]
```

- Find the numeric roots of $x^2 - 3$:

Input:

```
proot([1,0,-3])
```

or:

```
proot(x^2-3)
```

Output:

```
[-1.73205080757, 1.73205080757]
```


10.7 Numeric factorization of a matrix: cholesky qr lu svd

Matrix numeric factorizations of

- Cholesky,
- QR,
- LU,
- svd,

are described in Section 6.49 p.564.

Chapter 11

Unit objects and physical constants

11.1 Unit objects

11.1.1 Notation of unit objects

A unit object has two parts: a real number and a unit expression (a single unit or a multiplicative combination of units). The two parts are linked by the character `_` ("underscore"). For example `2_m` for 2 meters. For composite units, parenthesis must be used, e.g. `1_(m*s)`. See table 10.1 for a list of the basic units.

If a prefix is put before the unit then the unit is multiplied by a power of 10. For example, the prefix `k` or `K`, for kilo, indicates multiplication by 10^3 . See table 11.2 for a list of the unit prefixes. You cannot use a prefix with a built-in unit if the result gives another built-in unit.

For example: `1_a` is one are, but `1_Pa` is one pascal and not 10^{15}_a .

Examples.

- *Input:*

`10.5_m`

Output:

10.5 m

which is a unit object of value 10.5 meters.

- *Input:*

`10.5_km`

Output:

10.5 km

which is a unit object of value 10.5 kilometers.

Name	Description	Name	Description
<code>_A</code>	Ampere	<code>_ha</code>	Hectare
<code>_Angstrom</code>	Angstrom	<code>_hp</code>	Horsepower
<code>_Bq</code>	Becquerel	<code>_in</code>	Inch
<code>_Btu</code>	Btu British thermal unit	<code>_inH2O</code>	Inches of water, 60 degrees Fahrenheit
<code>_Ci</code>	Curie	<code>_inHg</code>	Inches of mercury, 0 degree Celsius
<code>_F</code>	Farad	<code>_j</code>	Day
<code>_Fdy</code>	Faraday	<code>_kWh</code>	Kilowatt-hour
<code>_Gal</code>	Gal (0.01 m/s ²)	<code>_kg</code>	Kilogram
<code>_Gy</code>	Gray	<code>_kip</code>	Kilopound-force
<code>_H</code>	Henry	<code>_km</code>	Kilometre
<code>_Hz</code>	Hertz	<code>_knot</code>	nautical miles per hour
<code>_J</code>	Joule	<code>_kph</code>	Kilometers per hour
<code>_K</code>	Kelvin	<code>_l</code>	Liter
<code>_Kcal</code>	Kilocalorie	<code>_lam</code>	Lambert
<code>_MHz</code>	Megahertz	<code>_lb</code>	pound (1 pound = 16 oz)
<code>_MW</code>	Megawatt	<code>_lbf</code>	Pound-force
<code>_MeV</code>	Megaelectronvolt	<code>_lbmol</code>	Pound-mole
<code>_N</code>	Newton	<code>_lbt</code>	Troy pound
<code>_Ohm</code>	Ohm	<code>_lep</code>	Liter of oil equivalent
<code>_P</code>	Poise (measures viscosity)	<code>_liqpt</code>	US liquid pint (1 US gallon = 8 US liquid pints)
<code>_Pa</code>	Pascal	<code>_lm</code>	Lumen
<code>_R</code>	Roentgen	<code>_lx</code>	Lux
<code>_Rankine</code>	Degree Rankine	<code>_lyr</code>	Light year
<code>_S</code>	Siemens	<code>_m</code>	Metre (unit)
<code>_St</code>	Stokes	<code>_mho</code>	Mho
<code>_Sv</code>	Sievert	<code>_miUS</code>	US statute mile
<code>_T</code>	Tesla	<code>_mi^2</code>	Square international mile.
<code>_V</code>	Volt	<code>_mil</code>	Mil
<code>_W</code>	Watt	<code>_mile</code>	International mile
<code>_Wb</code>	Weber	<code>_mille</code>	Nautical mile
<code>_Wh</code>	Watt-hour	<code>_ml</code>	millilitre
<code>_a</code>	Are (100 m ²)	<code>_mm</code>	Millimetre
<code>_acre</code>	Acre	<code>_mmHg</code>	Millimeter of mercury (torr), 0 degree Celsius
<code>_arcmin</code>	Minute of arc	<code>_mn</code>	Minute
<code>_arcs</code>	Second of arc	<code>_mol</code>	Mole
<code>_atm</code>	Atmosphere	<code>_mph</code>	Miles per hour
<code>_au</code>	Astronomical unit	<code>_oz</code>	Ounce
<code>_b</code>	Barn	<code>_ozUK</code>	UK fluid ounce
<code>_bar</code>	Bar	<code>_ozfl</code>	US fluid ounce
<code>_bbl</code>	Barrel	<code>_ozt</code>	Troy ounce
<code>_bblep</code>	Barrel of oil equivalent	<code>_pc</code>	Parsec
<code>_bu</code>	Bushel (1 bushel=8 gallons UK)	<code>_pdl</code>	Poundal (force)
<code>_buUS</code>	American bushel	<code>_ph</code>	Phot
<code>_cal</code>	Calorie	<code>_pk</code>	US peck
<code>_cd</code>	Candela	<code>_psi</code>	Pounds per square inch
<code>_chain</code>	Chain (1 chain = 66 feet or 22 yards)	<code>_ptUK</code>	UK pint (1 UK gallon=8 UK pints)
<code>_cm</code>	Centimetre	<code>_qt</code>	Quart
<code>_ct</code>	Carat	<code>_rad</code>	Radian
<code>_cu</code>	US cup	<code>_rd</code>	Rad (1 rd=0.01 Gy)
<code>_d</code>	Day	<code>_rem</code>	Rem
<code>_dB</code>	Decibel	<code>_rod</code>	Rod 1_rod=5.029215842_m
<code>_deg</code>	Degree (angle)	<code>_rpm</code>	Revolutions per minute
<code>_degreeF</code>	Degree Fahrenheit	<code>_s</code>	Second
<code>_dyn</code>	Dyne	<code>_s</code>	second
<code>_eV</code>	Electron volt	<code>_sb</code>	Stilb
<code>_erg</code>	Erg	<code>_slug</code>	Slug
<code>_fath</code>	Fathom	<code>_sr</code>	Steradian
<code>_fbm</code>	Board foot	<code>_st</code>	Stere
<code>_fc</code>	Footcandle (1 footcandle \approx 10.764 lux)	<code>_t</code>	Metric ton
<code>_fermi</code>	Fermi	<code>_tbsp</code>	Tablespoon
<code>_flam</code>	Footlambert	<code>_tec</code>	Tonne of coal equivalent
<code>_fm</code>	Fathom	<code>_tep</code>	Tonne of oil equivalent
<code>_ft</code>	International foot	<code>_tex</code>	tex=10 ⁻⁶ (kg/m)
<code>_ftUS</code>	Survey foot	<code>_therm</code>	EEC therm
<code>_g</code>	Gram	<code>_ton</code>	Short ton (1 short ton = 2000 pounds)
<code>_ga</code>	Standard freefall	<code>_tonUK</code>	Long (UK) ton
<code>_galC</code>	Canadian gallon	<code>_torr</code>	Torr (mmHg)
<code>_galUK</code>	UK gallon	<code>_tr</code>	tour=2 π rad
<code>_galUS</code>	US gallon	<code>_tsp</code>	Teaspoon
<code>_gf</code>	Gram-force	<code>_u</code>	Atomic mass unit
<code>_gmol</code>	Gram-mole	<code>_yd</code>	International yard
<code>_gon</code>	Grade	<code>_yr</code>	Year
<code>_grad</code>	Grade	<code>_μ</code>	Micron
<code>_grain</code>	Grain (1 grain \approx 0,0648 grams)	<code>_Åμ</code>	Micron
<code>_h</code>	Hour		

Table 11.1: Units

Prefix	Name	($\cdot 10^n$)	Prefix	Name	($\cdot 10^n$)
Y	yota	24	d	deci	-1
Z	zeta	21	c	cent	-2
E	exa	18	m	mili	-3
P	peta	15	mu	micro	-6
T	tera	12	n	nano	-9
G	giga	9	p	pico	-12
M	mega	6	f	femto	-15
k or K	kilo	3	a	atto	-18
h or H	hecto	2	z	zepto	-21
D	deca	1	y	yocto	-24

Table 11.2: Unit prefixes

11.1.2 Computing with units

Xcas performs usual arithmetic operations (+, -, *, /, ^) on unit objects. Different units may be used, but for + and - they must be compatible. The result is an unit object

- For the multiplication and the division of two unit objects $_u1$ and $_u2$, the unit of the result is written $_(u1 * u2)$ and $_(u1/u2)$.
- For an addition or a subtraction of compatible unit objects, the result is expressed with the same unit as the first term of the operation.

Examples.

- *Input:*

`1_m+100_cm`

Output:

2.0 m

- *Input:*

`100_cm+1_m`

Output:

200.0 cm

- *Input:*

`1_m*100_cm`

Output:

100 cmm

11.1.3 Converting units into MKSA units: `mksa`

The MKSA units are a system of units based on the meter, kilogram, second and ampere and usually used in scientific work. The `mksa` converts a unit object into a unit object written with the compatible MKSA base unit.

- `mksa` takes one argument:
 u , a unit object.
- `mksa(u)` returns the unit object in terms of the MKSA units.

Example.

Input:

```
mksa(15_C)
```

Output:

```
15.0 sA
```

11.1.4 Converting units: `convert =>`

The `convert` command (see Section 6.23.26 p.323) can convert a unit object into another compatible unit. For this:

- `convert` takes two arguments:
 - $unitobj$, a unit object.
 - u , a unit compatible with $unitobj$.
- `convert(unitobj,u)` returns $unitobj$ in terms of u .

Recall that the `=>` operator is the infix version of `convert`.

Examples.

- *Input:*

```
convert(1_h,_s)
```

Output:

```
3600.0 s
```

- *Input:*

```
convert(3600_s,_h)
```

Output:

```
1.0 h
```

11.1.5 Converting between Celsius and Fahrenheit: Celsius2Fahrenheit Fahrenheit2Celsius

The `Celsius2Fahrenheit` command converts a temperature in degrees Celsius to the equivalent temperature in Fahrenheit.

- *Celsius2Fahrenheit* takes one argument:
 T , a number representing representing a temperature in degrees Celsius.
- `Celsius2Fahrenheit(T)` returns the number representing the temperature in Fahrenheit.

Examples.

- *Input:*

`Celsius2Fahrenheit(a)`

Output:

$$\frac{9}{5}a + 32$$

- *Input:*

`Celsius2Fahrenheit(0)`

Output:

32

The `Fahrenheit2Celsius` command converts Fahrenheit temperatures to Celsius.

- *Fahrenheit2Celsius* takes one argument:
 T , a number representing representing a temperature in degrees Fahrenheit.
- `Fahrenheit2Celsius(T)` returns the number representing the temperature in Celsius.

Example.

Input:

`Fahrenheit2Celsius(212)`

Output:

100

11.1.6 Factoring a unit: `ufactor`

The `ufactor` command factors units in unit objects.

- `ufactor` takes two arguments:
 - *unitobj*, a unit object.
 - *u*, the unit to factor.
- `ufactor(unitobj,u)` returns a unit object multiplied by the remaining MKSA units.

Examples.

- *Input:*

```
ufactor(3_J,_W)
```

Output:

3.0 Ws

- *Input:*

```
ufactor(3_W,_J)
```

Output:

3.0 Js⁻¹

11.1.7 Simplifying units: `usimplify`

The `usimplify` command simplifies a unit in an unit object.

- `usimplify` takes one argument:
 - unitobj*, a unit object.
- `usimplify(unitobj)` returns *unitobj* with the units simplified.

Example.

Input:

```
usimplify(3_(W*s))
```

Output:

3.0 J

<code>_F_</code>	Faraday constant	<code>_h_</code>	Planck's constant
<code>_G_</code>	Gravitational constant	<code>_hbar_</code>	Dirac's constant
<code>_I0_</code>	Reference intensity	<code>_k_</code>	Boltzmann constant.
<code>_NA_</code>	Avogadro's number	<code>_kq_</code>	k/q (Boltzmann/charge of the electron)
<code>_PSun_</code>	Power at the surface of the Sun	<code>_lambda0_</code>	Photon wavelength (ch/e)
<code>_REarth_</code>	Radius of the Earth	<code>_lambdac_</code>	Compton wavelength
<code>_RSun_</code>	Radius of the Sun	<code>_mEarth_</code>	Mass of the Earth
<code>_R_</code>	Constante universelle des gaz	<code>_me_</code>	Electron rest mass
<code>_Rinfinity_</code>	Rydberg constant	<code>_mp_</code>	Proton rest mass
<code>_StdP_</code>	Standard pressure	<code>_mpme_</code>	Quotient mp/me (mass of the proton/mass of the electron)
<code>_StdT_</code>	Standard temperature	<code>_mu0_</code>	Permeability of vacuum
<code>_Vm_</code>	Molar volume	<code>_muB_</code>	Bohr magneton
<code>_a0_</code>	Bohr radius	<code>_muN_</code>	Nuclear magneton
<code>_alpha_</code>	Fine structure constant	<code>_phi_</code>	Magnetic flux quantum
<code>_angl_</code>	180 degree angle	<code>_q_</code>	Charge of an electron
<code>_c3_</code>	Wien displacement constant	<code>_qe_</code>	Electron charge
<code>_c_</code>	Speed of light in vacuum	<code>_qepsilon0_</code>	q*epsilon0 (charge of the electron*permittivity)
<code>_epsilon0_</code>	Permittivity of vacuum	<code>_qme_</code>	Quotient q/me (charge/mass of the electron)
<code>_epsilon0q_</code>	epsilon0/q (permittivity/charge of the electron)	<code>_rad_</code>	1 radian
<code>_epsilonox_</code>	Dielectric constant of Silicon dioxide	<code>_sd_</code>	Duration of a sidereal day
<code>_epsiloninsi_</code>	Dielectric constant	<code>_sigma_</code>	Stefan-Boltzmann constant
<code>_f0_</code>	Photon frequency (e/h)	<code>_syr_</code>	Duration of a sidereal year
<code>_g_</code>	Acceleration of gravity	<code>_twopi_</code>	2π

Table 11.3: Physical constants

11.2 Constants

11.2.1 Notation of physical constants

If you want to use a physical constants inside Xcas, put its name between two characters `_` ("underscore"). Don't confuse physical constants with symbolic constants; for example, e, π are symbolic constants and `_c_, _NA_` are physical constants. The physical constants are in the **Phys** menu, **Constant** sub-menu, and table 11.3 gives the Constants Library:

Examples.

- *Input:*

`_c_`

Output:

1 c

which represents the speed of light in vacuum. You can use the **mksa** command (see Section 11.1.3 p.830) to put this in terms of standard units:

Input:

`mksa(_c_)`

Output:

299792458.0 ms⁻¹

- *Input:*

`_NA_`

Output:

1 N_A

which represents Avogadro's number:

Input:

```
mksa(_NA_)
```

Output:

$6.0221367 \times 10^{23} \text{ mol}^{-1}$

Chapter 12

Programming

A program that you write for **Xcas** might be longer than one line; the first section discusses how you can enter it.

12.1 Functions, programs and scripts

12.1.1 The program editor

Xcas provides you with a program editor, which you can open with **Alt+P**. This can be useful for writing small programs, but for writing larger programs you may want to use your usual editor. (Note that this requires an editor, such as **emacs**, and not a word processor.) If you use your own editor, then you will need to save the program to a file, such as **myprog.cxx**, and then load it into **Xcas** with the command line command **load**:

Input:

```
load("myprog.cxx")
```

12.1.2 Functions: `function` `endfunction` `{ }` `local` `return`

You have already seen functions defined with `:=`. For example, to define a function **sumprod** which takes two inputs and returns a list with the sum and the product of the inputs, you can enter:

Input:

```
sumprod(a,b) := [a+b,a*b]
```

Afterwards, you can use this new function. *Input:*

```
sumprod(3,5)
```

Output:

```
[8,15]
```

You can define functions that are computed with a sequence of instructions by putting the instructions between braces, where each command ends with a semicolon. To use local variables, you can declare them with the **local** keyword, followed by the variable names. The value returned by the function will be indicated with the **return** keyword. For example, the above

function `sumprod` could also be defined by:

```
sumprod(a,b):= {
  local s, p;
  s:= a + b;
  p:= a*b;
  return [s,p];
}
```

Another way to use a sequence of instructions to define a function is with the `function ...endfunction` construction. With this approach, the function name and parameters follow the `function` keyword. This is otherwise like the previous approach. The `sumprod` function could be defined by:

Input:

```
function sumprod(a,b)
  local s, p;
  s:= a + b;
  p:= a*b;
  return [s,p];
endfunction
```

12.1.3 Local variables

Local variables in a function definition can be given initial values in the line they are declared in by putting their initialization in parentheses; for example,

```
local a,b;
a:= 1;
```

is the same as

```
local (a:= 1), b;
```

Local variables should be given values within the function definition. If you want to use a local variable as a symbolic variable, then you can indicate that with the `assume` command (see Section 5.4.8 p.104). For example, if you define a function `myroots` by

```
myroots (a):= {
  local x;
  return solve(x^2=a,x);
}
```

then calling

```
myroots(4)
```

will simply return the empty list. You could leave `x` undeclared, but that would make `x` a global variable and could interact with other functions in unexpected ways. You can get the behavior you probably expected by explicitly assuming `x` to be a symbol;

```
myroots (a):= {
  local x;
  assume(x,symbol);
  return solve(x^2=a,x);
}
```

(Alternatively, you could use `purge(x)` instead of `assume(x,symbol)`.) Now if you enter

```
myroots(4)
```

you will get

```
[-2,2]
```

12.1.4 Default values of the parameters

You can give the parameters of a function default values by putting *parameter=value* in the parameter list of the function. For example, if you define a function:

Input:

```
f(x,y=5,z):= {
  return x*y*z;
}
```

then:

Input:

```
f(1,2,3)
```

Output:

```
6
```

since the product $1 * 2 * 3 = 6$. If you give `f` only two values as input:

Input:

```
f(3,4)
```

Output:

```
60
```

since the values 3 and 4 will be given to the parameters which don't have default values; in this case, `y` will get its default value 5 while 3 and 4 will be assigned to `x` and `z`, respectively. The result is $x*y*z = 3 * 5 * 4 = 60$.

12.1.5 Programs

A program is similar to a function, and is written like a function without a return value. Programs are used to display results or to create drawings. It is a good idea to turn a program into a function by putting `return 0` at the end; this way you will get a response of 0 when the program executes.

12.1.6 Scripts

A script is a file containing a sequence of instructions, each ending with a semicolon.

12.1.7 Code blocks

A code block, such as used in defining functions, is a sequence of statements delimited by braces or by `begin` and `end`. Each statement must end with a semicolon. If the block makes up a function, you can step through it one statement at a time by using the debugger (see Section 12.5 p.867).

12.2 Basic instructions

12.2.1 Comments: //

The characters `//` indicate that you are writing a comment; any text between `//` and the end of the line will be ignored by Xcas.

12.2.2 Input: `input` `Input` `InputStr` `textInput` `output` `Output`

The `input` command prompts the user for the value of a variable. `Input` is a synonym for `input`.

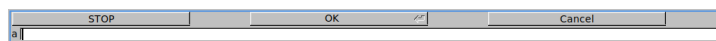
- `input` takes an unspecified number of commands:
`vars`, a sequence of variable names, each one optionally preceded by a string.
- `input(vars)` brings up a box where the user can enter a value for each variable.
 If a variable is preceded with a string, then that string will be the prompt for the variable, otherwise the variable name will be the prompt.

Examples.

- *Input:*

```
input(a)
```

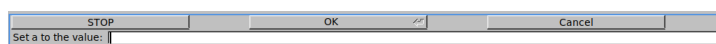
Output:



- *Input:*

```
input("Set a to the value: ",a)
```

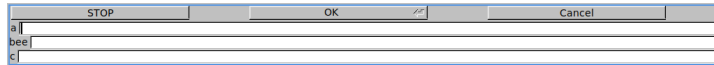
Output:



- *Input:*

```
input(a,bee,c)
```

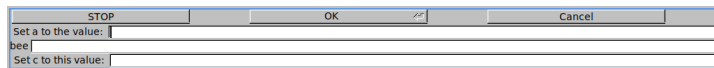
Output:



- *Input:*

```
input("Set a to the value: ",a,bee,"Set c to this value: ",c)
```

Output:



If the value that you enter for `input` is a string, it should be between quotes. If you want the user to enter a string without having to use the quotes, you can use the `InputStr` command, which is just like `input` except that it will assume any input is a string and so the user won't need to use quotes.

`textinput` is a synonym for `InputStr`.

The `output` command creates message windows:

`Output` is a synonym for `output`.

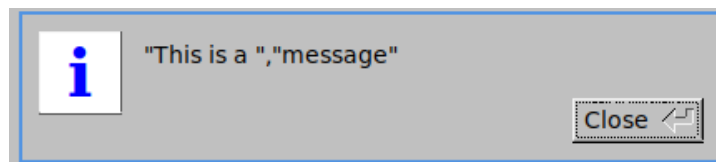
- `output` takes one argument:
`strs`, a sequence of strings or variables which represent strings.
- `output(strs)` creates a message window displaying the concatenation of the strings.

Example.

Input:

```
s := "message"
output("This is a ",s)
```

Output:



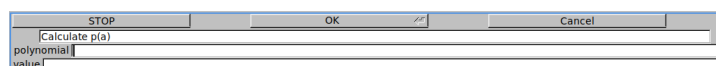
You can use `output` to add information to the input window.

Example.

Input:

```
input(output("Calculate p(a)"), "polynomial", p, "value", a)
```

Output:



12.2.3 Reading a single keystroke: getKey

The `getKey` command gets the next keystroke.

- `getKey` takes no arguments.
- `getKey()` returns the ASCII code of the next keystroke.

For example, if you enter

```
asciicode:= getKey()
```

and hit the A key, then the variable `asciicode` will have the value 65, which is the ASCII code of capital A.

12.2.4 Checking conditions: assert

The `assert` command breaks out of a function with an error.

- `assert` takes one argument:
bool, a boolean.
- `assert(bool)` does nothing if *bool* is true, it returns from the function with an error if *bool* is false.

Example.

Define the function:

Input:

```
sqofpos(x):= {assert(x > 0); return x^2;}
```

then:

```
sqofpos(4)
```

Output:

16

Input:

```
sqofpos(-4)
```

Output:

```
assert failure:  x>0 Error:  Bad Argument Value
```

since `-4 > 0` is false.

12.2.5 Checking the type of the argument: `type` `subtype` `compare` `getType`

The `type` command finds the type of its input.

- `type` takes one argument: *arg*.
- `type(arg)` returns an integer indicating the type of *arg*.
The integer is given as a constant symbol which is equal to the integer.
The possible values are:
 - 1, equivalently `real`, `double` or `DOM_FLOAT`.
 - 2, equivalently `integer` or `DOM_INT`.
 - 4, equivalently `complex` or `DOM_COMPLEX`.
 - 6, equivalently `identifier` or `DOM_IDENT`.
 - 7, equivalently `vector` or `DOM_LIST`.
 - 8, equivalently `expression` or `DOM_SYMBOLIC`.
 - 10, equivalently `rational` or `DOM_RAT`.
 - 12, equivalently `string` or `DOM_STRING`.
 - 13, equivalently `func` or `DOM_FUNC`.

Examples.

- *Input:*

```
type(4)
```

Output:

```
integer
```

- *Input:*

```
type(3.1) == DOM_FLOAT
```

Output:

```
true
```

`Xcas` has various types of lists; the `subtype` command can determine what kind of list it is.

- `subtype` takes one argument:
L, a list (in `DOM_LIST`).
- `subtype(L)` returns an integer indicating what type of list *L* is.
The possible values are:
 - 1 is *L* is a sequence.

- 2 if L is a set.
- 10 if L is a polynomial represented by a list (see Section 6.27 p.351).
- 0 if L isn't one of the above types of list.

Example.*Input:*`subtype(1,2,3)`*Output:*

1

The `compare` operator compares two objects taking their type into account.

- `compare` takes two arguments:
 a, b , two objects.
- `compare(a, b)` returns
 - 1 (`true`) if `type(a) < type(b)` or if `type(a) = type(b)` and a is less than b in the ordering of their type.
 - 0 (`false`) otherwise.

Examples.

- *Input:*

`compare("a","b")`*Output:*

1

since "a" and "b" have the same type (`string`) and "a" is less than "b" in the string ordering.

- If `b` is a formal variable:

Input:`compare("a",b)`*Output:*

0

since the type of "a" is `string` (the integer 12) while the type of `b` is `identifier` (the integer 6) and 12 is not less than 6.

The `getType` command is similar to `type` in that it takes an object and returns the type, but it has different possible return values. It is included for compatibility reasons.

- `getType` takes one argument:
obj, an object.
- `getType(obj)` returns the type of *obj*, which in this case means one of:
NUM, VAR, STR, EXPR, NONE, PIC, MAT or FUNC.

Examples.

- *Input:*

```
getType(3.14)
```

Output:

```
NUM
```

- *Input:*

```
getType(x)
```

Output:

```
VAR
```

12.2.6 Printing: print Disp ClrIO

The `print` command prints in a special pane.

`Disp` is a synonym for `print`.

- `print` takes one argument:
seq, a sequence of objects.
- `print(seq)` returns 1 and prints the *seq* in a special pane just above the output line.

Examples.

- *Input:*

```
print("Hello")
```

Output:

```
Hello  
1
```

- *Input:*

```
a:= 12  
print("a =",a)
```

Output:

```
a =,12
1
```

The `ClrIO` command erases printing on the level it was typed.

- `ClrIO` takes no arguments and no parentheses.
- `ClrIO` returns 1 and erases any printing on the special pane above the output line on the level it was typed.

Example.

Input:

```
print("Hello"); ClrIO
```

Output:

```
(1,1)
```

12.2.7 Displaying exponents: `printpow`

The `printpow` command determines how the `print` command will print exponents in the special pane above the output line.

- `printpow` takes on argument:
 n , either -1,0 or 1 (by default 1).
- `printpow(n)` sets the style for printing exponents with the `print` command.
 - If $n = -1$, `print(a^b)` will subsequently print `$a**b$` . a^b .
 - If $n = 0$, `print(a^b)` will subsequently print `$\text{pow}(a,b)$` .
 - If $n = 1$, `print(a^b)` will subsequently print a^b .

Example.

Input:

```
print(x^3)
```

Above the output line:

```
x^3
```

Input:

```
printpow(-1)
print(x^3)
```

Above the output line:

```
a**b
```

Input:

```
printpow(0)
print(x^3)
```

Above the output line:

```
pow(a,b)
```

Input:

```
printpow(1)
print(x^3)
```

Above the output line:

```
print(x^3)
```

12.2.8 Infix assignments: => := =<

The infix operators =>, :=, and =< can all store a value in a variable, but their arguments are in different order. (See Section 5.4.2 p.100 and Section 5.4.3 p.101.) Also, := and =< have different effects when the first argument is an element of a list stored in a variable, since =< modifies list elements by reference (see section 12.2.10).

- => is the infix version of `sto`, it stores the value in the first argument in the variable in the second argument. Both

```
4 => a
```

and:

```
sto(4,a)
```

store the value 4 in the variable `a`.

- := and =< both have a variable as the first argument and the value to store in the variable as the second argument. Both

```
a:= 4
```

and:

```
a =< 4
```

store the value 4 in the variable `a`.

However, suppose you have entered:

```
A:= [0,1,2,3,4]
B:= A
```

and you want to change `A[3]`.

```
A[3] =< 33
```

will change both `A` and `B`:

Input:

```
A, B
```

Output:

```
[0, 1, 2, 33, 4], [0, 1, 2, 33, 4]
```

Here, `A` pointed to the list `[0,1,2,3,4]` and after setting `B` to `A`, `B` also pointed to `[0,1,2,3,4]`. Changing an element of `A` by reference changes the list that `A` points to, which `B` also points to.

Note that multiple assignments can be made using sequences or lists. Both

```
[a, b, c] := [1, 2, 3]
```

and:

```
(a, b, c) := (1, 2, 3)
```

assign `a` the value 1, `b` the value 2, and `c` the value 3. If multiple assignments are made this way and variables are on the right hand side, they will be replaced by their values before the assignment. If `a` contains 5 and you enter:

```
(a,b) := (2,a)
```

then `b` will get the previous value of `a`, 5, and not the new value of `a`, 2.

12.2.9 Assignment by copying: `copy`

The `copy` command creates a copy of its argument, which is typically a list of some type. If `B` is a list and `A := B`, then `A` and `B` point to the same list, and so changing one will change the other. But if `A := copy(B)`, then `A` and `B` will point to different lists with the same values, and so can be changed individually.

Example.

Input:

```
B:= [[4,5], [2,6]]
A:= B
C:= copy(B)
A, B, C
```

Output:

```
⎡ 4 5 ⎤ ⎡ 4 5 ⎤ ⎡ 4 5 ⎤
⎣ 2 6 ⎦ ⎣ 2 6 ⎦ ⎣ 2 6 ⎦
```

Input:

```
B[1] =< [0,0]
A, B, C
```

Output:

$$\begin{bmatrix} 4 & 5 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} 4 & 5 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} 4 & 5 \\ 2 & 6 \end{bmatrix}$$

12.2.10 The difference between := and =<

The := and =< assignment operators have different effects when they are used to modify an element of a list contained in a variable, since =< modifies the element by reference. Otherwise, they will have the same effect.

Example.

Input:

```
A:= [1,2,3]
```

Here:

```
A[1] := 5
```

and

```
A[1] =< 5
```

both change A[1] to 5:

Input:

```
A
```

Output:

```
[1,5,3]
```

but they do it in different ways. The command A[1] =< 5 changes the middle value in the list that A originally pointed to, and so any other variable pointing to the list will be changed, but A[1] := 5 will create a duplicate list with the middle element of 5, and so any other variable pointing to the original list won't be affected.

Examples.

- *Input:*

```
A:=[0,1,2,3,4]
B:=A
B[3]=<33
A,B
```

Output:

```
[0,1,2,33,4], [0,1,2,33,4]
```

- *Input:*

```
A:=[0,1,2,3,4] B:=A B[3]:=33 A,B
```

Output:

`[0, 1, 2, 3, 4] , [0, 1, 2, 33, 4]`

If B is set equal to a copy of A instead of A, then changing B won't affect A.

Example.

Input:

`A:=[0,1,2,3,4] B:=copy(A) B[3]=<33 A,B`

Output:

`[0, 1, 2, 3, 4] , [0, 1, 2, 33, 4]`

12.3 Control structures

12.3.1 if statements: if then else end elif

The Xcas language has different ways of writing `if...then` statements (see Section 6.1.3 p.114). The standard version of the `if...then` statement consists of the `if` keyword, followed by a boolean expression (see Section 6.1 p.113) in parentheses, followed by a statement block (see Section 12.1.7 p.838) which will be executed if the boolean is true. You can optionally add an `else` keyword followed by a statement block which will be executed if the boolean is false:

`if (boolean) true-block {else false-block}`

(where recall the blocks need to be delimited by braces or by `begin` and `end`).

Examples.

• *Input:*

```
a:=3; b:=2;;
if (a > b) { a:= a + 5; b:= a - b;};;
a,b
```

Output:

8,6

since `a > b` will evaluate to true, and so the variable `a` will be reset to 8 and `b` will be reset to the value 6.

• *Input:*

```
a:=3; b:=2;; if (a < b) { a:= a + 5; b:= a - b;} else { a:=a - 5; b:= a + b}
```

Output:

-2,0

since `a > b` will evaluate to false, and so the variable `a` will be reset to -2 and `b` will be reset to the value 0.

An alternate way to write an `if` statement is to enclose the code block in `then` and `end` instead of braces:

```
if (boolean) then true-block {else false-block} end
```

Examples.

- *Input:*

```
a := 3
if (a > 1) then a:= a + 5; end
```

Output:

8

- *Input:*

```
a:=8
if (a > 10) then a:= a + 10; else a:= a - 5; end
```

Output:

3

This input can also be written:

```
si (a > 10) alors a:= a + 10; sinon a:= a - 5; fsi
```

Several `if` statements can be nested; for example, the statement

```
if (a > 1) then a:= 1; else if (a < 0) then a:= 0; else a:=
    0.5; end; end
```

A simpler way is to replace the `else if` by `elif` and combine the `ends`; the above statement can be written

```
if (a > 1) then a:= 1; elif (a < 0) then a:= 0; else a:= 0.5;
    end
```

In general, such a combination can be written

```
if (boolean 1) then
    block 1;
elif (boolean 2) then
    block 2;
...
elif (boolean n) then
    block n;
else
    last block;
end
```

(where the last `else` is optional.) For example, if you want to define a function f by

$$f(x) = \begin{cases} 8 & \text{if } x > 8 \\ 4 & \text{if } 4 < x \leq 8 \\ 2 & \text{if } 2 < x \leq 4 \\ 1 & \text{if } 0 < x \leq 2 \\ 0 & \text{if } x \leq 0 \end{cases}$$

you can enter

```
f(x):= {
  if (x > 8) then
    return 8;
  elif (x > 4) then
    return 4;
  elif (x > 2) then
    return 2;
  elif (x > 0) then
    return 1;
  else
    return 0;
  end;
}
```

12.3.2 The switch statement: switch case default

The **switch** statement can be used when you want the value of a block to depend on an integer. It takes one argument, an expression which evaluates to an integer. It should be followed by a sequence of **case** statements, which takes the form **case** followed by an integer and then a colon, which is followed by a code block to be executed if the expression equals the integer. At the end is an optional **default:** statement, which is followed by a code block to be executed if the expression doesn't equal any of the given integers:

```
switch(n) {
  case n1:  block n1
  case n2:  block n2
  ...
  case nk:  block nk
  default:  default_block
```

(where recall the blocks need to be delimited by braces or by **begin** and **end**).

Example.

As an example of a program which performs an operation on the first two variables depending on the third, you could enter (see Section 12.1.1 p.835):

```

oper(a,b,c):= {
switch (c) {
  case 1:  {a:= a + b; break;}
  case 2:  {a:= a - b; break;}
  case 3:  {a:= a * b; break;}
  default: {a:= a ^ b;}
}
return a;
}

```

Then:

Input:

`oper(2,3,1)`

Output:

5

since the third argument is 1, and so `oper(a,b,c)` will return $a + b$, and:

Input:

`oper(2,3,2)`

Output:

-1

since the third argument is 2 and so `oper(a,b,c)` will return $a-b$.

12.3.3 The for loop: for from to step do end_for

The **for** loop has three different forms, each of which uses an index variable. If the **for** loop is used in a program, the index variable should be declared as a local variable. (Recall that i represents the imaginary unit, and so cannot be used as the index.)

The first form: For the first form, the **for** is followed by the starting value for the index, the end condition, and the increment step, separated by semicolons and in parentheses. Afterwards is a block of code to be executed for each iteration:

`for (j:=j0; end_cond; increment_step) block`

where j is the index and j_0 is the starting value of j .

Example.

To add the even numbers less than 100, you can start by setting the running total to 0:

Input:

`S:= 0`

then use a **for** loop to do the summing:

Input:

`for (j:= 0; j < 100; j:= j + 2) {S:= S + j}`

Output:

2450

The second form: The second form of a `for` loop has a fixed increment for the index. It is written out with `for` followed by the index, followed by `from`, the initial value, `to`, the ending value, `step`, the size of the increment, and finally the statements to be executed between `do` and `end_for`:

```
for j from j0 to jmax step k do statements end_for
```

where j is the index, j_0 is the initial value of j , j_{max} is the ending value of j , k is the step size of j , and *statements* are executed for each value of j .

Example.

Again, to add the even numbers less than 100, you can start by setting the running total to 0:

Input:

```
S:= 0
```

then use the second form of the `for` loop to do the summing:

Input:

```
for j from 2 to 98 step 2 do S:= S + j; end_for
```

or (a French version of this syntax):

```
pour j de 2 jusque 98 pas 2 faire S:= S + j; fpour
```

Output:

```
2450
```

The third form: The third form of the `for` loop lets you iterate over the values in a list (or a set or a range). In this form, the `for` is followed by the index, then `in`, the list, and then the instructions between `do` and `end_for`:

```
for j in L do statements end_for
```

where j is the index and L is the list to iterate over.

Example.

To add all integers from 1 to 100, you can again set the running total S to 0:

Input:

```
S:=0
```

then use the third form of the `for` loop to add the integers:

Input:

```
for j in 1..100 do S:= S + j; end_for
```

or:

```
pour j in 1..100 faire S:= S + j; fpour
```

Output:

```
5050
```

12.3.4 The repeat loop: repeat until

The **repeat** loop allows you to repeat statements until a given condition is met. To use it, enter **repeat**, the statements, the keyword **until** followed by the condition, a boolean:

```
repeat statements until bool
```

Example.

If you want the user to enter a value for a variable **x** which is greater than 4, you could use:

```
repeat
input("Enter a value for x (greater than 4)",x);
until (x > 4);
```

This can also be written

```
repeter
input("Enter a value for x (greater than 4)",x);
jusqua (x > 4);
```

12.3.5 The while loop: while

The **while** loop is used to repeat a code block as long as a given condition holds. To use it, enter **while**, the condition in parentheses, and then a code block.

```
while (bool) block
```

Example.

Add the terms of the harmonic series $1 + 1/2 + 1/3 + 1/4 + \dots$ until a term is less than 0.05.

You can initialize the sum **S** to 0 and let **j** be the first term 1.

Input:

```
S:=0
j:=1
```

Then use a while loop:

Input:

```
while (1/j >= 0.05) {S:= S + 1/j; j:= j+1;}
```

or:

```
tantque (1/j >= 0.05) faire S:= S + 1/j; j:= j+1; ftantque
```

then:

```
S
```

Output:

```
55835135
15519504
```

Note that a **while** loop can also be written as a **for** loop. For example, as long as **S** is set to 0 and **j** is set to 1, the above loop can be written as

```
for (;1/j >= 0.05;) {S:= S + 1/j; j:= j+1;}
```

or, with only `S` set to 0,

```
for (j:= 1; 1/j >= 0.05; j++) {S:= S + 1/j;}
```

12.3.6 Breaking out a loop: `break`

The `break` command exits a loop without finishing it.

- `break` takes no arguments or parentheses.
- `break` exits the current loop.

Example.

Define a program:

```
testbreak(a,b):= {
  local r;
  while (true) {
    if (b == 0) {break;}
    r:= irem(a,b);
    a:= b;
    b:= r;
  }
  return a;
}
```

Then:

Input:

```
testbreak(4,0)
```

Output:

4

since the `while` loop is interrupted when `b` is 0 and `a` is 4.

12.3.7 Going to the next iteration of a loop: `continue`

The `continue` command will skip the rest of the current iteration of a loop and go to the next iteration.

- `continue` takes no arguments or parentheses.
- `continue` goes to the next iteration of the current loop without finishing the current iteration.

Example.

If you enter:

```

S:= 0
for (j:= 1, j <= 10; j++) {
  if (j == 5) {continue;}
  S:= S + j;
}

```

then S will be 50, which is the sum of the integers from 1 to 10 except for 5, since the loop never gets to $S := S + j$ when j is equal to 5.

12.3.8 Changing the order of execution: goto label

The `goto` command will tell a program to jump to a different spot in a program, where the spot needs to have been marked with `label`. They both must have the same argument, which is simply a sequence of characters.

- `label` takes one argument:
mark, a sequence of characters.
- `label(mark)` labels the position in the program with *mark*.
- `goto` takes one argument:
mark, a sequence of characters.
- `goto(mark)` goes to the part of the program labeled with *mark*.

Example.

The following program will add the terms of the harmonic series until the term is less than some specified value `eps` and print the result.

```

harmsum(eps):= {
  local S, j;
  S:= 0;
  j:= 0;
  label(spot);
  j:= j + 1;
  S:= S + 1/j;
  if (1/j >= eps) goto (spot);
  print(S);
  return 0;
}

```

12.4 Other useful instructions

12.4.1 Defining a function with a variable number of arguments: args

The `args` command returns the list of arguments of a function.

- `args` takes no arguments.
- `args` (or `args(NULL)`) returns a list of the arguments of the current function, starting with the name of the function at index 0.

Note that `args()` will not work, the command must be called as `args` or `args(NULL)`. You can also use `(args)[0]` to get the name of the function and `(args)[1]` to get the first argument, etc., but the parentheses about `args` is mandatory.

Examples.

- *Input:*

```
testargs() := {local y; y:= args; return y[1];}
testargs(12,5)
```

Output:

12

- Enter the function:

```
total() := {
  local s,a;}
a:=args;
s:=0;
for (k:=1;k<size(a);k++){
  s:=s+a[k];
}
return s;
}
```

then:

```
total(1,2,3,4)
```

Output:

10

12.4.2 Assignments in a program

Recall that the `=<` operator will change the value of a single entry in a list or matrix by reference (see Section 5.4.3 p.101). This make it efficient when changing many values, one at a time, in a list, as might be done by a program.

You must be careful when doing this, since your intent might be changed when a program is compiled. For example, if a program contains

```
local a;
a:= [0,1,2,3,4];
...
a[3] =< 33;
```

then in the compiled program, `a:= [0,1,2,3,4]` will be replaced by `a:= [0,1,2,33,4]`. To avoid this, you can assign a copy of the list to `a`; you could write:


```

local a;
a:= copy([0,1,2,3,4]);
...
a[3] =< 33;

```

Alternately, you could use a command which recreates a list every time the program is run, such as `makelist` or `$`, instead of copying a list; `a:=makelist(n,n,0,4)` or `a:= [n$(n=0..4)]` can also be used in place of `a:= [0,1,2,3,4]`.

12.4.3 Writing variable values to a file: write

The `write` command saves variable values to a file, to be read later.

- `write` takes two arguments:
 - *filename*, a string.
 - *vars*, a sequence of variables.
- `write(filename,vars)` writes the variables in *vars* assigned to their values in a file named *filename*.

Example.

Input:

```

a:=3.14
b:=7
write("foo",a,b)

```

creates a file named “foo” containing:

```

a:=(3.14);
b:=7;

```

If you wanted to store the first million digits of π to a file, you could set it equal to a variable and store it in a file: *Input:*

```

pidec:= evalf(pi,10^6)::
write("pilmillion",pidec)

```

The file is written so that it can be loaded with the `read` command (Section 5.6.2 p.111), which simply takes a file name as a string. This allows you to restore the values of variables saved this way, for example in a different session or if you have purged the variables.

Example.

If, in a different session, you want to use the values of `a` and `b` above, you can enter:

Input:

```

read("foo")

```

This will reassign the values 3.14 and 7 to `a` and `b`. Be careful, this will silently overwrite any values that `a` and `b` might have had.

12.4.4 Writing output to a file: `fopen` `fclose` `fprint`

You can use the `fopen`, `fprint` and `fclose` commands to write output to a file instead of the screen.

The `fopen` command creates and opens a file to write into.

- `fopen` takes one argument:
filename, a string.
- `fopen(filename)` creates a file named *filename* (and erases it if it already exists).

To use this, you need to associate it with a variable `var:=fopen(filename)` which you can use to refer to the file when printing to it.

The `fprint` command writes to a file.

- `fprint` takes two mandatory arguments and one optional argument:
 - *var*, a variable name associated with a file through `fopen`.
 - Optionally, `Unquoted`, the symbol.
 - *info*, a list of what you want to write to the file.
- `fprint(var < Unquote >, info)` writes *info* into the file given by *var*. By default, strings in *info* are written with their quotation marks, with the option `Unquoted`, `fprint` will print them with the quotation marks.

The `fclose` command closes a file.

- `fclose` takes one argument:
var, a variable assigned to a file with `fopen`.
- `fclose(var)` closes the file given by *var* to further writing.

Example.

To write contents to a file, you first need to open the file and associate it with a variable.

Input:

```
f:= fopen("bar")
```

This creates a file named “bar” (and so erase it if it already exists). To write to the file:

Input:

```
x:=9
fprint(f,"x + 1 is ", x+1)
```

will put

```
"x + 1 is "10
```

in the file. Note that the quotation marks are inserted with the `Unquoted` argument:

Input:

```
x:=9
fprintf(f,Unquoted,"x + 1 is ", x+1)
```

will put

```
x + 1 is 10
```

in the file. Finally, after you have finished writing what you want into the file, you close the file with the `fclose` command:

Input:

```
fclose(f)
```

12.4.5 Using strings as names: `make_symbol`

Variable and function names are symbols, namely sequences of characters, which are different from strings. For example, you can have a variable named `abc`, but not `"abc"`. The `make_symbol` command turns a string into a symbol; for example `make_symbol("abc")` is the symbol `abc`.

Examples.

- *Input:*

```
a:= "abc";
make_symbol(a) := 3
```

or:

```
make_symbol("abc") := 3
```

then:

```
abc
```

Output:

```
3
```

The variable `abc` will have the value 3.

- Similarly for functions.

Input:

```
b:= "sin";
make_symbol(b)(pi/4)
```

or:

```
make_symbol("sin")(pi/4)
```

Output:

$$\frac{\sqrt{2}}{2}$$

which is `sin(pi/4)`.

12.4.6 Using strings as commands: `expr`

The `expr` command lets you use a string as a command.

- `expr` takes one argument:
`str`, a string which expresses a valid command.
- `expr(str)` converts `str` to the command and evaluates it.

Examples.

- *Input:*

```
expr("c:= 1")
c
```

Output:

1

- *Input:*

```
a:= "ifactor(54)"
expr(a)
```

Output:

$2 \cdot 3^3$

which is the same thing as entering `ifactor(54)` directly.

You can also use `expr` to convert a string to a number. If a string is simply a number enclosed by quotation marks, then `expr` will return the number.

Example.

Input:

```
expr("123")
```

Output:

123

In particular, the following strings will be converted to the appropriate number.

- A string consisting of the digits 0 through 9 which doesn't start with 0 will be converted to an integer.

Example.

Input:

```
expr("2133")
```

Output:

2133

- A string consisting of the digits 0 through 9 which contains a single decimal point will be converted to a decimal.

Example.*Input:*

```
expr("123.4")
```

Output:

```
123.4
```

- A string consisting of the digits 0 through 9, possibly containing a single decimal point, followed by `e` and then more digits 0 through 9, will be read as a decimal in exponential notation.

Example.*Input:*

```
expr("1.23e4")
```

Output:

```
12300.0
```

- A string consisting of the digits 0 through 7 which starts with 0 will be read as an integer base 8.

Example.*Input:*

```
expr("0176")
```

Output:

```
126
```

since 176 base 8 equals 126 base 10.

- A string starting with `0x` followed by digits 0 through 9 and letters `a` through `f` will be read as an integer base 16.

Example.*Input:*

```
expr("0x2a3f")
```

Output:

```
10815
```

since 2a3f base 16 equals 10815 base 10.

- A string starting with `0b` followed by digits 0 and 1 will be read as a binary integer.

Example.*Input:*

```
expr("0b1101")
```

Output:

```
13
```

since 1101 base 2 equals 13 base 10.

12.4.7 Converting an expression to a string: `string`

The `string` command converts an expression to a string.

- `string` takes one argument:
expr, an expression.
- `string(expr)` evaluates *expr* then converts it to a string.

Example.

Input:

```
string(ifacto(6))
```

Output:

```
"2 * 3"
```

This is the same thing as adding the empty string to the expression:

```
ifacto(6) + ""
```

If you want to convert an unevaluated expression to a string, you can quote the expression (see Section 6.12.4 p.205).

Example.

Input:

```
string(quote(ifacto(6)))
```

```
"ifacto(6)"
```

12.4.8 Converting a real number into a string: `format`

The `format` command converts real numbers into strings.

- `format` takes two arguments:
 - *r*, a real number.
 - *str*, a string used for formatting.
- `format(str)` returns *r* as a string with the requested formatting.

The formatting string can be one of the following:

- `f` (for *floating* format) followed by the number of digits to put after the decimal point.

Example.

Input:

```
format(sqrt(2)*10^10,"f13")
```

Output:

```
"14142135623.7308959960938"
```

- **s** (for *scientific* format) followed by the number of significant digits.

Example.

Input:

```
format(sqrt(2)*10^10,"s13")
```

Output:

```
"14142135623.73"
```

- **e** (for *engineering* format) followed by the number of digits to put after the decimal point, with one digit before the decimal points.

Example.

Input:

```
format(sqrt(2)*10^10,"e13")
```

Output:

```
"1.4142135623731e+10"
```

12.4.9 Working with the graphics screen: DispG DispHome ClrGraph ClrDraw

Recall that the **DispG** screen contains the graphical output of **Xcas**. The **DispG** command opens the **DispG** screen.

- **Disp** takes no arguments and no parentheses.
- **Disp** brings up the **Disp** screen.

Example.

Input:

```
DispG;
```

opens the graphics screen.

The **ClrGraph** command clears the screen.

ClrDraw is a synonym for **ClrGraph**.

- **ClrGraph** takes no arguments.
- **ClrGraph** clears the **Disp** screen.

Example.*Input:*

```
ClrGraph
```

or:

```
ClrGraph()
```

erases the **DispG** screen.

The **DispHome** command closes the **DispG** screen.

- **DispHome** takes no arguments and no parentheses.
- **DispHome** closes the **DispG** screen.

Example.*Input:*

```
DispHome;
```

makes the graphics screen go away.

12.4.10 Pausing a program: Pause WAIT

The **Pause** command pauses **Xcas**.

- **Pause** takes one optional argument (with no parentheses). Optionally, r , a positive number.
- **Pause** r pauses **Xcas** for r seconds.
- **Pause** brings up a **Pause** informational window and pauses **Xcas** until you click **Close** in the **Pause** window.

Example.*Input:*

```
Pause 10
```

pauses **Xcas** for 10 seconds.

The **WAIT** command also pauses **Xcas**. It acts just like **Pause**, but uses parentheses for its argument.

Example.*Input:*

```
WAIT(10)
```

pauses **Xcas** for 10 seconds.

12.4.11 Dealing with errors: try catch throw error ERROR

Some commands produce errors, and if your program tries to run such a command it will halt with an error. The `try` and `catch` commands help you avoid this. They to use them, put potentially problematic statements in a block following `try`, and immediately after the block put `catch` with an argument of an unused symbol, and follow that with a block of statements that can deal with the error.

```
try tryblock catch symbol catchblock
```

If *tryblock* doesn't produce an error, then

```
catch symbol catchblock
```

If *tryblock* does produce an error, then a string describing the error is assigned to *symbol*, and *catchblock* is evaluated.

Examples.

- The command

```
[[1,1]]*[[2,2]]
```

produces an error saying *Error: Invalid dimension*. However,

```
try {[[1,1]]*[[2,2]]}
catch (err) {
  print("The error is " + err)
}
```

will not produce an error:

Output:

```
The error is Error: Invalid dimension
```

- With the following program:

```
test(x) := {
  local y, str, err; try { y:= [[1,1]]*x; str:= "This produced a product.";}
  catch (err)
  {y:= x;
   str:= "This produced an error " + err + " The input is returned.";}
  print(str);
  return y;
}
```

Input:

```
test([[2],[2]])
```

Output:

```
This produced a product.
[4]
```

with the text in the pane above the output line.

Input:

```
test([[2,2]])
```

Output:

```
This produced an error Error: Invalid dimension The input is returned.
[[2,2]]
```

with the text in the pane above the output line.

You can produce your own string to describe an error message with the **throw** command.

error and **ERROR** are synonyms for **throw**.

- **throw** takes one argument:
 str, a string describing an error.
- **throw(str)** generates an error with error string *str*, possibly to be caught by **catch**.

Example.

With the program:

```
f(x) := {
  if (type(x) != DOM_INT)
    throw("Not an integer");
  else
    return x;
}
```

Input:

```
f(12)
```

Output:

```
12
```

since 12 is an integer.

Input:

```
f(1.2)
```

will signal an error

Not an integer Error: Bad Argument Value

since 1.2 is not an integer.

You can catch this error in other programs. Consider the program:

```
g(x) := {
  try(f(x)) catch(err) {x:= 0;}
  return x;
}
```

then:

Input:

g(12)

Output:

12

since 12 is an integer.

Input:

g(1.2)

Output:

0

since 1.2 is not an integer, `f(x)` will give an error and so `g(x)` will return 0.

12.5 Debugging

12.5.1 Starting the debugger: `debug sst in sst_in cont`
`kill break breakpoint halt rmbrok rmbreakpoint watch rmwch`

The `debug` command starts the Xcas debugger.

- `debug` takes one argument:
`fn(arg)`, a function and its argument.
- `debug(fn(arg))` brings up a debug window which contains a pane with the program with the current line highlighted, an `eval` entry box, a pane with the program including the breakpoints, a row of buttons, and a pane keeping track of the values of variables.

By default, the value of all variables in the program are in this pane. The buttons are shortcuts for entering commands in the `eval` box, but you can enter other commands in the `eval` box to change the values of variables or to run a command in the context of the program.

Example.

With the `sumprod` program:

```

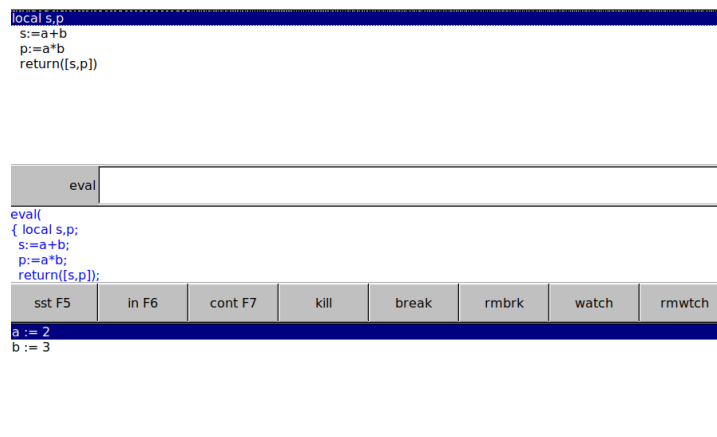
sumprod(a,b) := {
  local s, p;
  s:= a + b;
  p:= a*b;
  return [s,p];
}

```

Input:

```
debug(sumprod(2,3))
```

Output:



The debug window has the following buttons:

- sst** This button will run the `sst` command, which takes no arguments and runs the highlighted line in the program before moving to the next line.
- in** This button will run the `sst_in` command, which takes no argument and runs one step in the program or a user defined function used in the program.
- cont** This button will run the `cont` command, which takes no arguments and runs the commands from the highlighted line to a breakpoint.
- kill** This button will run the `kill` command, which exits the debugger.
- break** This button will put the command `breakpoint` in the `eval` box, with default arguments of the current program and the current line. It sets a breakpoint at the given line of the given program. Alternatively, if you click on a line in the program in the top pane, you will get the `breakpoint` command with that program and the line you clicked on.

You can set a breakpoint when you write a program with the `halt()` command. When a program has a `halt` command, then running the program will bring up the debugger. If you want

to debug the program, though, it is still better to use the `debug` command. Also, you should remove any `halt` commands when you are done debugging.

rmbrok This button will put the command `rmbreakpoint` in the `eval` box, with default arguments of the current program and the current line. It removes a breakpoint at the given line of the given program. Alternatively, you can click on the line in the program in the top pane with the bookmark you want to remove.

watch This button will put the command `watch` in the `eval` box, without the arguments filled in. It takes a list of variables as arguments, and will keep track of the values of these variables in the variable pane.

rmwtch This button will put the command `rmwatch` in the `eval` box without the arguments filled in. The arguments are the variables you want to remove from the watch list.

Chapter 13

Two-dimensional Graphics

13.1 Introduction

13.1.1 Points, vectors and complex numbers

A point in the Cartesian plane is described with an ordered pair (a, b) . It has x -coordinate (abscissa) a and y -coordinate (ordinate) b .

A vector from one point (a_1, b_1) to another (a_2, b_2) has associated ordered pair $(a_2 - a_1, b_2 - b_1)$; so the abscissa is $a_2 - a_1$ and the ordinate is $b_2 - b_1$.

A complex number $a + bi$ can be associated with the point (a, b) in the Cartesian plane. The complex number is called the *affix* of the point.

A point in **Xcas** is specified with the **point** command (see Section [13.6.2](#) p.886), which takes as argument either two real numbers a, b or a complex number $a + bi$. In this chapter, when a command take a point as an argument, the point can either be the result of the **point** command or simply a complex number.

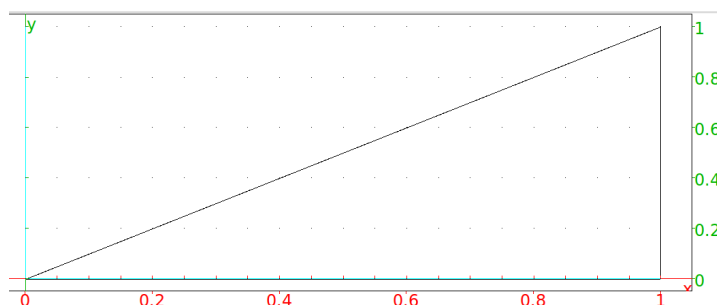
An interactive graphic screen opens whenever a geometric object is drawn, or with the command **Alt+G**. The objects on the screen can also be created and manipulated with the mouse.

As an example (to be explained in more detail later), the **triangle** command draws a triangle; the result will be a graphics screen containing axes, the triangle and a control panel on the right.

Input:

```
triangle(0,1,1+i)
```

Output:



13.2 Basic commands

13.2.1 Clearing the DispG screen: `erase`

The `DispG` screen records all graphic commands since the beginning of the session or the screen was last erased. The `Alt-D` command (or the menu command `Cfg ► Show ► DispG`) brings up this screen.

The `erase` command clears the `DispG` screen without restarting the session.

Input:

```
erase
```

or:

```
erase()
```

clears the `DispG` screen. This can be useful for commands such as `graph2tex`, which only takes into account the objects on the `DispG` screen.

13.2.2 Toggling the axes: `switch_axes`

The `switch_axes` command shows, hides or toggles the coordinate axes on the graphics screen. This can also be controlled by a `show axes` checkbox in the configuration panel brought up with the `cfg` button on the graphic screen control panel.

- `switch_axes` takes one optional argument:
 n , either 0 or 1.
- `switch_axes()` toggles whether or not the coordinate axes are show in subsequent graphics screens.
- `switch_axes(0)` causes all later graphic screens to omit the axes.
- `switch_axes(1)` causes all later graphic screens to have the axes.

When the axes are visible, they have tick marks whose separation is determined by the X-tick and Y-tick values on the graphic configuration screen. Setting these values to 0 also removes the axes.

13.2.3 Drawing unit vectors in the plane: `0x_2d_unit_vector` `0y_2d_unit_vector` `frame_2d`

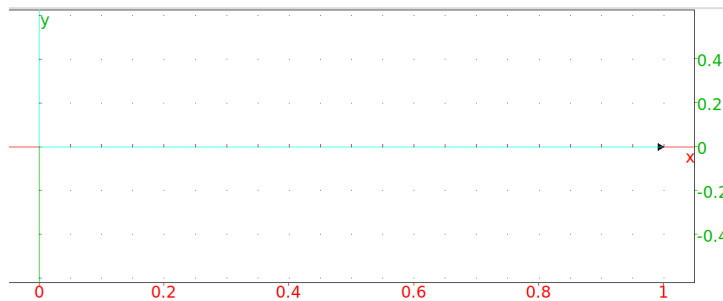
The `0x_2d_unit_vector` command takes no arguments and draws the unit vector in the x -direction on a plane.

Example.

Input:

```
0x_2d_unit_vector()
```

Output:



Similarly, the `0y_2d_unit_vector` command draws the unit vector in the y direction. The `frame_2d` command simultaneously draws both unit vectors.

13.2.4 Drawing dotted paper: `dot_paper`

The `dot_paper` command draws dotted paper.

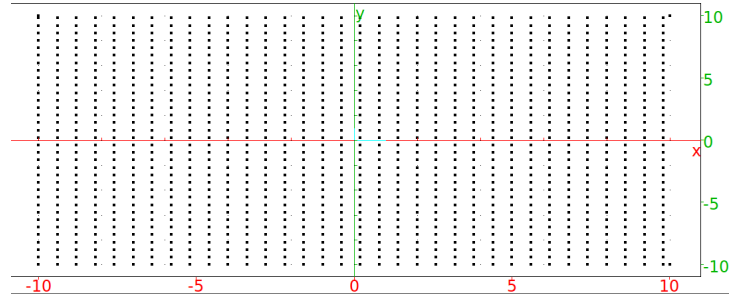
- `dot_paper` takes three mandatory arguments and two optional arguments.
 - * *xspacing*, the spacing in the x direction.
 - * θ , the angle from the horizontal to draw the dots.
 - * *yspacing*, the spacing in the y direction.
 - * Optionally, $x=x_{min}..x_{max}$, to determine how far the dots extend in the x direction (by default, the distances given in the graphic configuration page accessible from the main menu).
 - * Optionally, $y=y_{min}..y_{max}$, to determine how far the dots extend in the y direction (by default, the distances given in the graphic configuration page accessible from the main menu).
- `dot_paper(xspacing, θ ,yspacing [$x=x_{min}..x_{max}$, $y=y_{min}..y_{max}$])` draws the dotted paper.

Example.

Input:

```
dot_paper(0.6,pi/2,0.6)
```

Output:



Unchecking **Show Axes** on the **cfg** screen removes the axes.

13.2.5 Drawing lined paper: `line_paper`

The `line_paper` command draws lined paper.

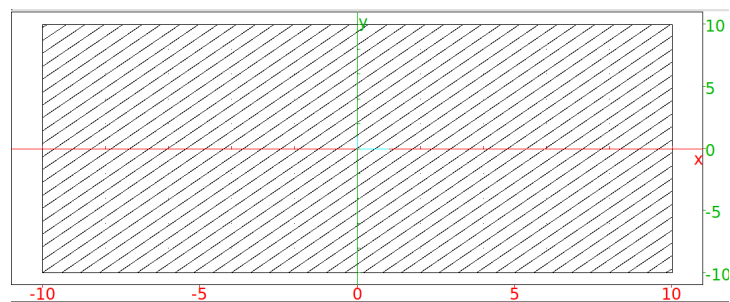
- `line_paper` takes two mandatory arguments and two optional arguments.
 - * *xspacing*, the spacing in the *x* direction.
 - * θ , the angle from the horizontal to draw the lines.
 - * Optionally, $x=x_{min}..x_{max}$, to determine how far the lines extend in the *x* direction (by default, the distances given in the graphic configuration page accessible from the main menu).
 - * Optionally, $y=y_{min}..y_{max}$, to determine how far the lines extend in the *y* direction (by default, the distances given in the graphic configuration page accessible from the main menu).
- `line_paper(xspacing, θ $\langle x=x_{min}..x_{max}, y=y_{min}..y_{max} \rangle$)` draws the lined paper.

Example.

Input:

```
line_paper(0.6, pi/3)
```

Output:



Unchecking **Show Axes** on the **cfg** screen removes the axes.

13.2.6 Drawing grid paper: `grid_paper`

The `grid_paper` command draws grid paper.

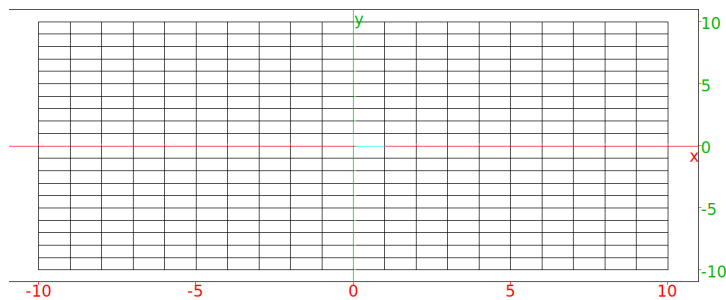
- `grid_paper` takes three mandatory arguments and two optional arguments.
 - * *xspacing*, the spacing in the *x* direction.
 - * θ , the angle from the horizontal to draw the grid.
 - * *yspacing*, the spacing in the *y* direction.
 - * Optionally, $x=x_{min}..x_{max}$, to determine how far the grid extends in the *x* direction (by default, the distances given in the graphic configuration page accessible from the main menu).
 - * Optionally, $y=y_{min}..y_{max}$, to determine how far the grid extends in the *y* direction (by default, the distances given in the graphic configuration page accessible from the main menu).
- `grid_paper(xspacing, θ ,yspacing $\langle x=x_{min}..x_{max},y=y_{min}..y_{max} \rangle$)` draws the grid paper.

Example.

Input:

```
grid_paper(1,pi/2,1)
```

Output:



Unchecking **Show Axes** on the `cfg` screen removes the axes.

13.2.7 Drawing triangular paper: `triangle_paper`

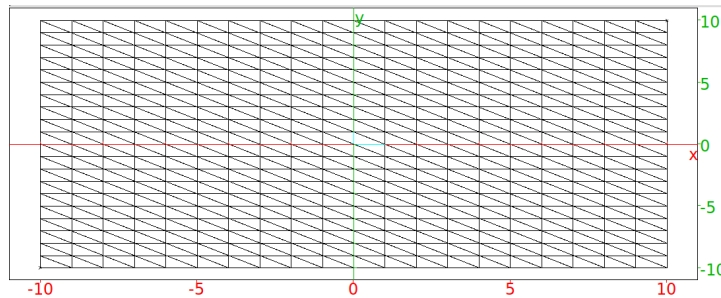
The `triangle_paper` command draws triangular paper.

- `triangle_paper` takes three mandatory arguments and two optional arguments.
 - * *xspacing*, the spacing in the *x* direction.
 - * θ , the angle from the horizontal.
 - * *yspacing*, the spacing in the *y* direction.
 - * Optionally, $x=x_{min}..x_{max}$, to determine how far the grid extends in the *x* direction (by default, the distances given in the graphic configuration page accessible from the main menu).

- * Optionally, $y=y_{min}..y_{max}$, to determine how far the grid extends in the y direction (by default, the distances given in the graphic configuration page accessible from the main menu).
- `triangle_paper(xspacing, θ ,yspacing $\langle x=x_{min}..x_{max},y=y_{min}..y_{max} \rangle$)` draws the triangle paper.

Example.*Input:*

```
triangle_paper(1,pi/2,1)
```

Output:

Unchecking **Show Axes** on the **cfg** screen removes the axes.

13.3 Display features of graphics

13.3.1 Graphic features

Graphic objects and graphic screens can have features, such as labels and colors, that are only included when requested, and other features, such as line width, which are configurable. Some features will be global, meaning that they will apply to the entire graphic screen, and some will be local, meaning that they will only apply to individual objects.

13.3.2 Parameters for changing features

Graphical features are changed by giving appropriate values to certain parameters. Several values can be given at once with an expression of the form *feature* = value1 + value2 + Some values can be set using optional arguments to graphic commands, which will set the feature locally; namely, it will only apply to that particular graphic object. Some values can be specified at the beginning of a line, which will set the feature globally; it will apply to all the graphic objects created on that line. For some features, both options are available.

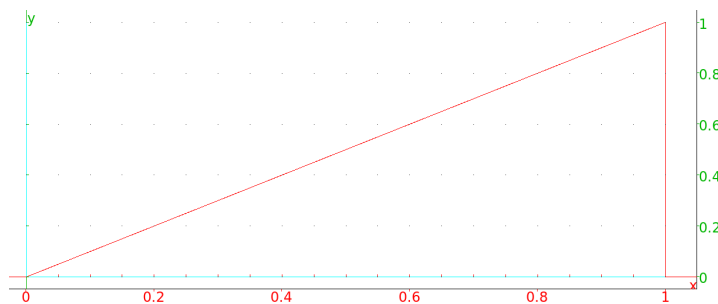
Parameters for local features

Commands which create graphic objects, such as `triangle`, can have optional arguments to change a features of the object. For example, the argument `color = red` will make an object red.

Input:

```
triangle(0,1,1+i,color=red)
```

Output:



The features and their possible values are:

display or color These two parameter names have the same effect. They control the following features.

Color The following values will change the color:

- An integer from 0 to 381.
Integers from 0 to 255 correspond to the color palette, integers from 256 to 381 will be the spectrum of colors. The program below will demonstrate the colors and their numbers.
- The names `black`, `white`, `red`, `blue`, `green`, `magenta`, `cyan` or `yellow`.

Fill The `filled` value creates a solid object.

Point markers By default, points are drawn with a small cross. The following (self-explanatory) values change the marker.

```
rhombus_point
square_point
cross_point
star_point
plus_point
point_point
triangle_point
invisible_point
```

Point width The values `point_width_1`, ..., `point_width_8` change the thickness of the lines in the point markers.

Line style The following (self-explanatory) values change the style of lines.

```

solid_line
dash_line
dashdot_line
dashdotdot_line
cap_flat_line
cap_round_line
cap_square_line

```

Line widths The values `line_width_1`, ..., `line_width_8` change the thickness of the lines.

thickness This controls line thickness, it can be an integer from 1 to 7.

nstep This sets the number of sampling points for three-dimensional objects.

tstep This sets the step size of the parameter when drawing a one parameter parametric plot.

ustep This sets the step size of the first parameter when drawing a two-parameter parametric plot.

vstep This sets the step size of the second parameter when drawing a two-parameter parametric plot.

xstep This sets the step size of the x variable.

ystep This sets the step size of the y variable.

zstep This sets the step size of the z variable.

frames This sets the number of graphs computed when an animated graph is created with the `animate` or `animate3d` command.

legend This adds a legend to a graphic object and should be a string. It is probably most useful when that object is a point or a polygon. If the object is a polygon, the legend will be placed in the middle of the last side. Other parameters for the graphic object will specify the color or position of the legend.

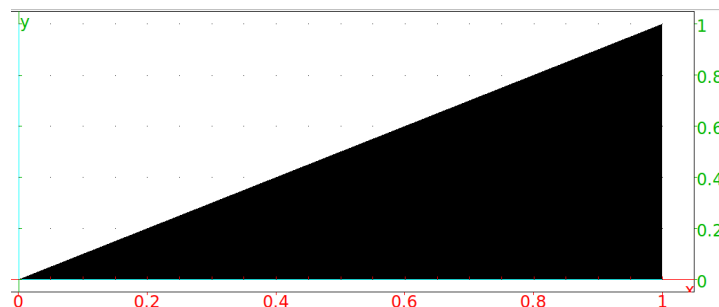
gl_texture This sets an image file to be put on the graphic object; it should be the name of the file.

Example (of the filled option)

Input:

```
triangle(0,1,1+i,display=filled)
```

Output:



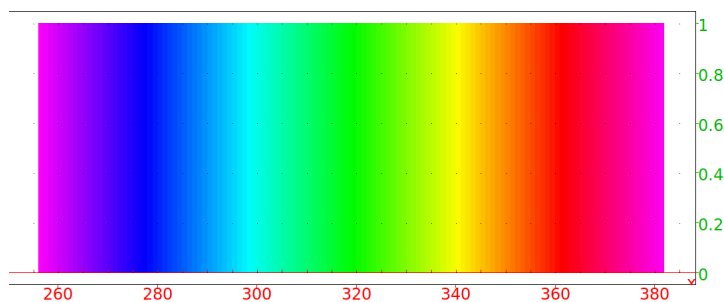
To see the colors the numbers can represent, you can run the program:

```
rainbow() := {
  local j, C;
  C := [];
  for (j := 256; j < 382; j++) {
    C := append(C, square(j, j+1, color=j+filled));
  }
}
```

Input:

```
rainbow();
```

Output:



The number of a color is its x -coordinate. To see just one color, say the color corresponding to n for $256 \leq n \leq 381$, enter:

Input:

```
rainbow()[n-256]
```

Parameters for global features

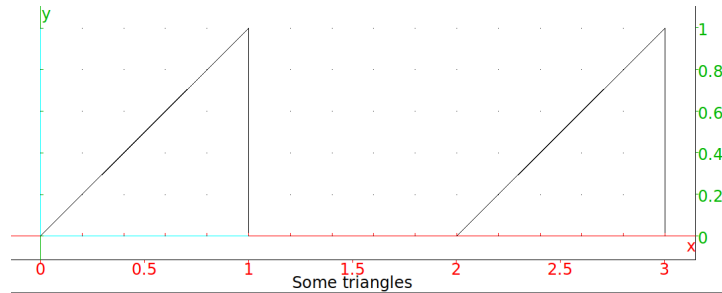
Parameters set at the beginning of a line change features on the entire graphic screen. It only takes effect when the line ends with a graphic command. For example, starting the line with `title=title string` will give the graphic screen a title.

Example.

Input:

```
title = "Some triangles"; triangle(0,1,1+i);
triangle(2,3,3+i);
```

Output:



The parameters for global features and their possible values are:

axes This determines whether axes are shown or hidden; a value of 0 or **false** hides the axes, a value of 1 or **true** shows the axes.

labels This sets labels for the axes; it should be a list of two strings `["x axis label", "y axis label"]`.

label This puts labels on the graphic screen in the following ways.

- To set the units on the axes, it can be a list of two or three strings, `["x units", "y units"]` or `["x units", "y units", "z units"]`.
- To place a string at a particular point, it can be a list of two integers followed by a string. The integers determine the point, starting from [0,0] in the top left of the screen.

title This sets the title for the graphic window, it should be a string.

gl_texture This sets the wallpaper of the graphic window to be an image file, it should be the name of the file.

gl_x_axis_name, gl_y_axis_name, gl_z_axis_name These set the names of the axes.

gl_x_axis_unit, gl_y_axis_unit, gl_z_axis_unit These set the units of the axes.

gl_x_axis_color, gl_y_axis_color, gl_z_axis_color These set the colors of the axes labels; they take the same color options as the local parameter **color**.

gl_ortho This ensures that the graph is orthonormal when it is set to 1.

gl_x, gl_y, gl_z These define the framing of the graph; they should be ranges *min..max*. (They are not compatible with interactive graphs.)

gl_xtick, gl_ytick, gl_ztick These determine the spacing of the ticks on the axes.

gl_shownames This shows or hides object names, it can be **true** or **false**.

gl_rotation This sets the axis of rotation for three-dimensional scene animations; it should be a direction vector $[x, y, z]$.

gl_quaternion This sets the quaternion for viewing three-dimensional scenes; it should be a fourtuple $[x, y, z, t]$. (This is not compatible with interactive graphs.)

13.3.3 Commands for global display features

Adding a legend: `legend`

The `legend` command creates a legend on the screen.

- `legend` takes two mandatory arguments and one optional argument:
 - * *pos*, either be a point or a list of two integers giving the number of pixels from the upper left hand corner, specifying the position to put the legend.
 - * *legend*, a string or a variable.
 - * Optionally, *quad*, which can be one of `quadrant1`, `quadrant2`, `quadrant3` or `quadrant4`. This indicates where to put the legend relative to the point (by default, it is `quadrant1`).
- `legend(pos, legend [quad])` draws the legend at the requested position.

Example.

To put "hello" to the upper left of the point (1,1):

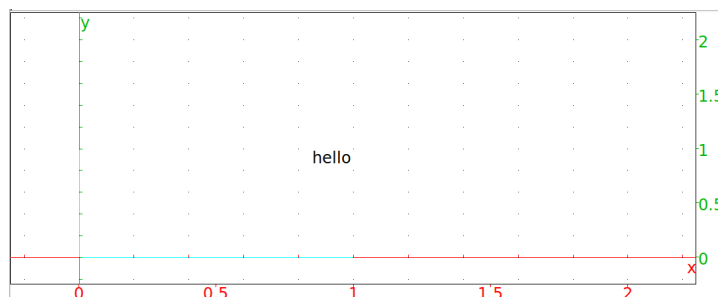
Input:

```
legend(1+i,"hello",quadrant3)
```

or:

```
legend(1+i,quadrant3,"hello")
```

Output:



Changing various features: `display color`

The `display` command changes the properties of graphics; the same properties that can also be changed with the `display` and `color` parameters (see Section 13.3.2 p.877). The `color` command a synonym for the `display` command.

The `display` command draws objects with specified properties.

- `display` takes one mandatory arguments and one optional argument:
 - * Optionally, *command*, a command to draw an object.
 - * *arg*, which can be a possible value of the `display` parameter (see Section 13.3.2 p.877) or `hidden_name`.
- `display(command,arg)` draws the object given by *command* with the property given by *arg*, or draws the object without a label if *arg*=`hidden_name`.
- `display(arg)` applies the property given by *arg* to all subsequent objects; `display(0)` resets the display parameters.

Examples.

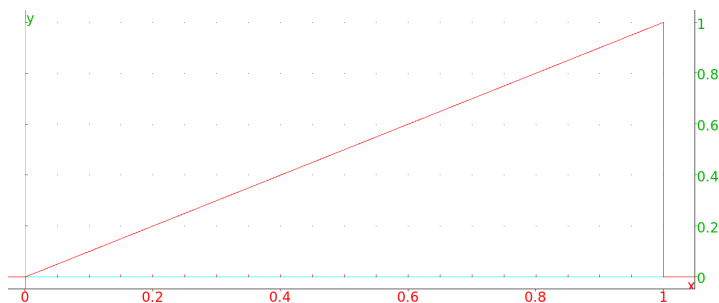
- *Input:*

```
display(triangle(0,1,1+i),red)
```

or:

```
triangle(0,1,1+i,display=red)
```

Output:



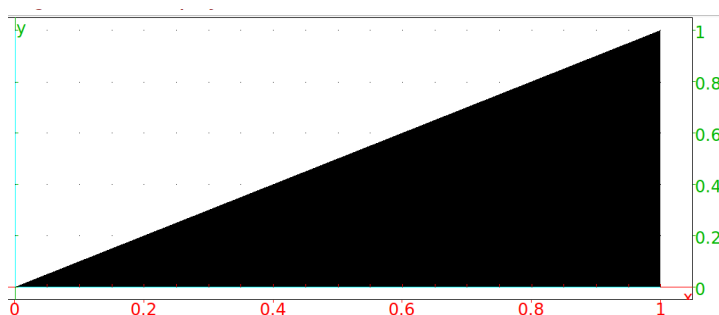
- *Input:*

```
triangle(0,1,1+i,display=filled)
```

or:

```
display(triangle(0,1,1+i),filled)
```

Output:

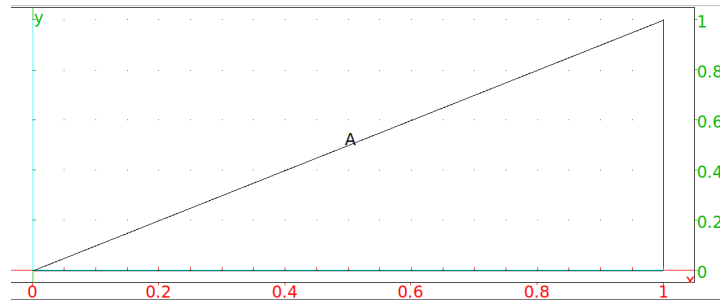


- By default, if a geometric object is named, the drawing is labeled.
Input:

13.4. DEFINING GEOMETRIC OBJECTS WITHOUT DRAWING THEM: `NODISP`

```
A:= triangle(0,1,1+i)
```

Output:

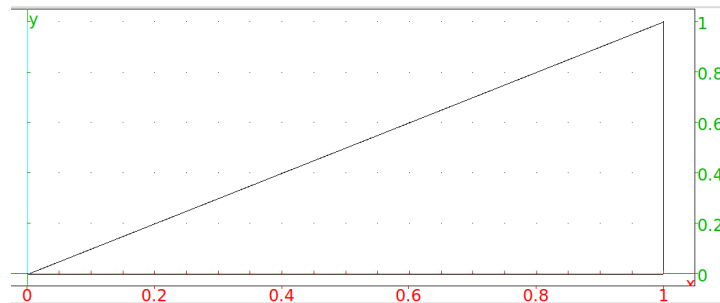


Creating the object with the `display` command and the `hidden_name` argument will draw it without the label.

Input:

```
display(A:= triangle(0,1,1+i),hidden_name)
```

Output:



13.4 Defining geometric objects without drawing them: `nodisp`

The `nodisp` command defines an object without displaying it.

- `nodisp` takes one argument:
command, a command to create an object.
- `nodisp(command)` creates the object without drawing it.

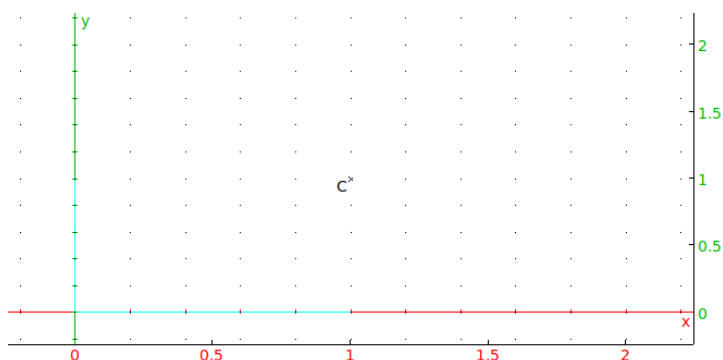
Setting a variable to a graphic object draws the object.

Examples.

– *Input:*

```
C:= point(1+i)
```

Output:



Input:

```
nodisp(C:= point(1+i))
```

Here, the point C is defined but not displayed. It is equivalent to following the command with `::`,

Input:

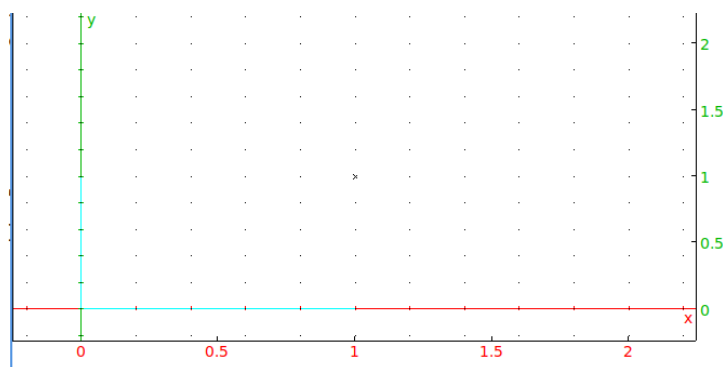
```
C:= point(1+i)::
```

To define a point as above and display it without the label, enter the point's name;

Input:

C

Output:



Alternatively, you can get the same effect by defining the point within an `eval` statement:

Input:

```
eval(C:= point(1+i))
```

To later display the point with a label, use the `legend` command:

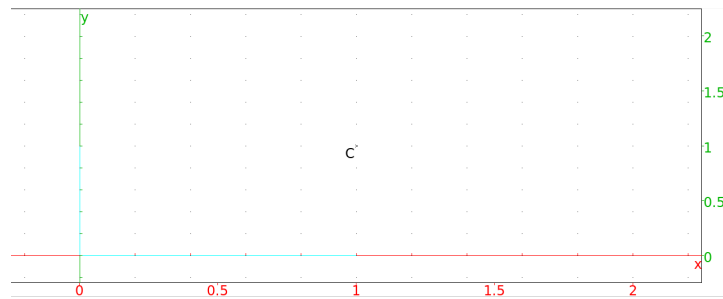
Input:

```
legend(C, "C")
```

or:

```
point(affix(C), legend="C")
```

Output:



In this case, the string "C" can be replaced with any other string as a label. Alternatively, redefine the variable as itself:

Input:

```
C:= C
```

prints C with its label.

13.5 Geometric demonstrations: assume

Variables should be unspecified to demonstrate a general geometric result, but need to have specific values when drawing. There are a couple of different approaches to deal with this.

One approach is to use the **assume** command (see Section 5.4.8 p.104). If a variable is *assumed* to have a value, then that value will be used in graphics but the variable will still be unspecified for calculations. For example: *Input:*

```
assume(a = 2.1)
A:= point(a + i)
```

Output:



but the variable a will still be treated as a variable in calculations:

Input:

```
distance(0,A)
```

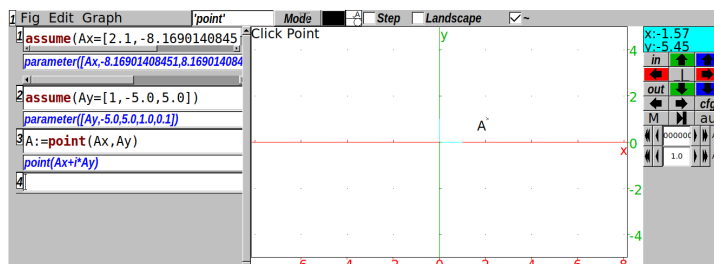
Output:

$$\sqrt{(-a)^2 + 1}$$

Another approach would be to use the `point` or `pointer` mode in a geometry screen. If there isn't a geometry screen showing, the command `Alt-G` or the `Geo►New figure 2d` menu will open a screen. Clicking on the `Mode` button right above the graphic screen and choosing `pointer` or `point` will put the screen in `pointer` or `point` mode. If a point is defined and displayed, such as with `A:= point(2.1 + i)`, then clicking on the name of the point (`A` in this case) with the right mouse button will bring up a configuration screen. As long as there is a point defined with non-symbolic values, there will be a `symp` box on the configuration screen. Selecting the `symp` box and choosing `OK` will be equivalent to the commands:

```
assume(Ax=[2.1, -8.16901408451, 8.16901408451])
assume(Ay = [1, -5.0, 5.0])
```

This will bring up two lines beneath the arrows to the right of the screen which can be used to change the assumed values of `Ax` and `Ay`. Also, the point `A` will be redefined as `point(Ax,Ay)`.



13.6 Points in the plane

13.6.1 Points and complex numbers

The *affix* of a point (a, b) in the plane is the complex number $a + bi$. In this section, when a command takes points as arguments, the points can be specified by a pair or by a complex number.

13.6.2 The point in the plane: `point`

See Section 14.4.1 p.987 for points in space.

In the 2-d geometry screen in point mode, clicking on a point with the left mouse button will choose that point. Points chosen this way are automatically named, first with `A`, then `B`, etc.

Alternatively, the `point` command chooses a point.

- `point` takes one or two arguments:
 - `coords`, where `coords` can be one of:
 - * a, b , a sequence of two coordinates.

- * $[a, b]$, a list of two coordinates.
- * $a + bi$, the affix of the point.
- `point(coords)` returns and draws the point with the given coordinates.

Example.*Input:*

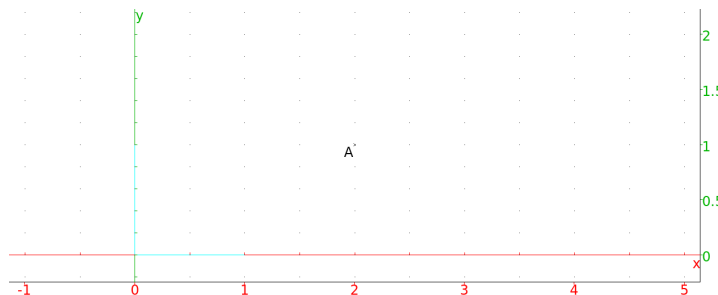
```
A:= point(2,1)
```

or:

```
A:= point([2,1])
```

or:

```
A:= point(2 + i)
```

Output:

The marker used to indicate the point can be changed; see Section [13.3.2](#) p.877.

If the `point` command has two numbers for arguments, at least one of which is complex but not real, then it will choose two points.

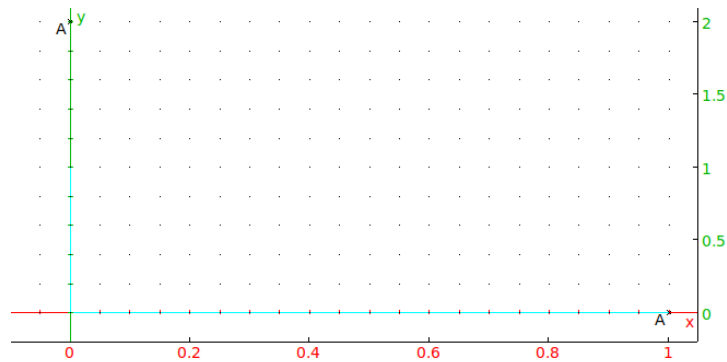
Example.*Input:*

```
A:= point(1,2*i)
```

or:

```
A:= point([1,2*i])
```

Output:



There are two points named A; one with affix 1 and one with affix $2i$.

13.6.3 The difference and sum of two points in the plane:+

–

Let A and B be two points in the plane, with affixes $a_1 + ia_2$ and $b_1 + ib_2$ respectively.

Input:

```
A:= point(1 + 2*i); B:= point(3+4*i)
```

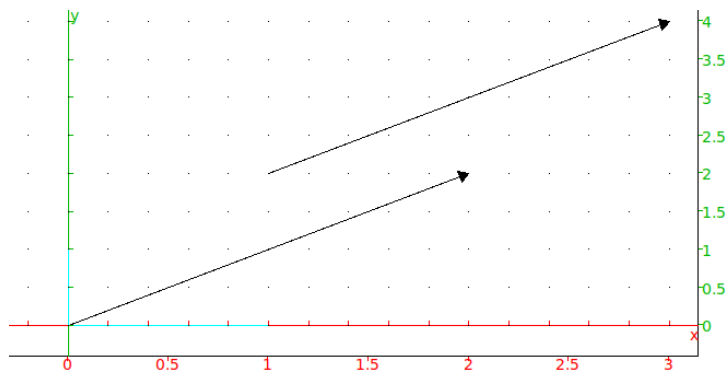
Then:

- The difference $B - A$ returns the affix $(b_1 - a_1) + i(b_2 - a_2)$, which represents the vector AB .

Input:

```
vector(A,B); vector(point(0),point(B-A))
```

Output:

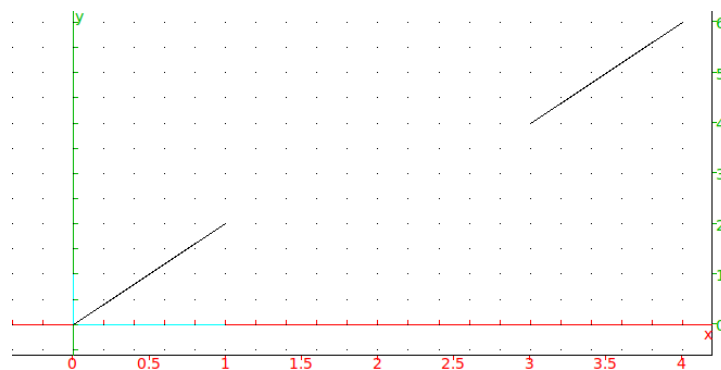


- The sum $B + A$ returns the affix $(b_1 + a_1) + i(b_2 + a_2)$. If $D := \text{point}(B+A)$, then $BD = OA$.

Input:

```
D:= point(B + A);
segment(B,D); segment(point(0),A)
```

Output:



Note that $-A$ is the point symmetrical to A with respect to the origin. The sum of three points $A + B + C$ can be viewed as the translate of C by the vector $A + B$. So if A or B contains parameters, you should write this as $C + (A + B)$ or $evalc(A + B) + C$.

13.6.4 Defining random points in the plane: `point2d`

The `point2d` command defines a random point whose coordinates are integers between -5 and 5.

- `point2d` takes an unspecified number of arguments:
`names`, a sequence of names for the points.
- `point2d(names)` assigns a random point whose coordinates are integers between -5 and 5 to each name.

Examples.

– *Input:*

```
point2d(A)
```

This assigns A to a random point. Once assigned, the point is fixed.

– *Input:*

```
point2d(A,B,C)
triangle(A,B,C)
```

generates three random points and uses them to create a triangle; i.e., it creates a random triangle.

13.6.5 Points in polar coordinates: `polar_point` `point_polar`

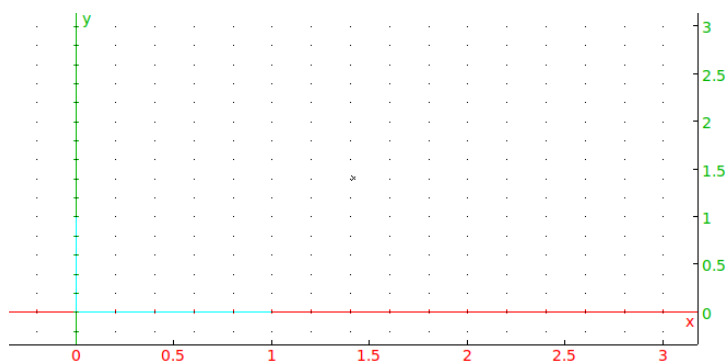
You can use the `point` command to specify a point in polar coordinates by using the polar representation of complex numbers.

Example.

Input:

```
point(2*exp(i*pi/4))
```

Output:



which is the point with polar coordinates $r = 2$, $\theta = \pi/4$.

The `polar_point` command is an easier way to specify a point in polar coordinates.

`point_polar` is a synonym for `polar_point`.

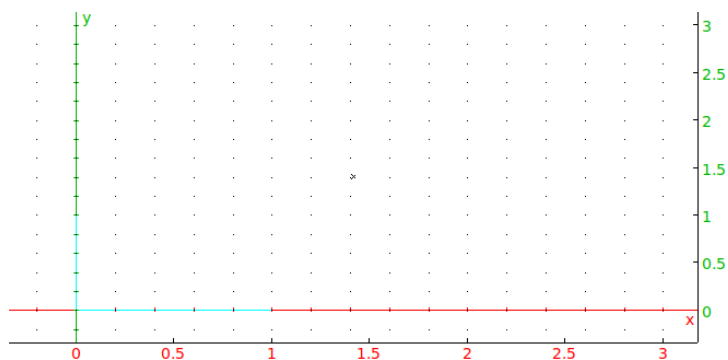
- `polar_point` takes two arguments:
 - * r , a number.
 - * θ , a number.
- `polar_plot(r, θ)` returns and draws the point with polar coordinates r, θ .

Example.

Input:

```
polar_point(2,pi/4)
```

Output:



which is the same point as before.

13.6.6 Finding a point of intersection of two objects in the plane: `single_inter` `line_inter`

See Section 14.4.3 p.989 for single points of intersection of objects in space.

The `single_inter` command finds an intersection point of two geometric objects.

`line_inter` is a synonym for `single_inter`

- `single_inter` takes two mandatory arguments and one optional argument.

- * obj_1, obj_2 , two geometric objects.

- * Optionally, pt , a point or list of points.

`line_inter(obj1,obj2⟨ pt ⟩)` returns one of the points of intersection of obj_1 and obj_2 .

If pt is a single point, then the command returns the point of intersection closest to pt .

If pt is a list of points, then the command tries to return a point not in pt .

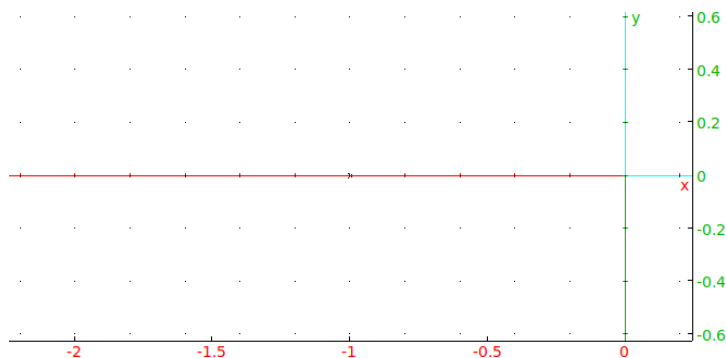
Example.

The command `circle(0,1)` creates the unit circle and `line(-1,i)` creates a line, these two objects intersect at the points $(-1,0)$ and $(0,1)$.

Input:

```
single_inter(circle(0,1),line(-1,i))
```

Output:

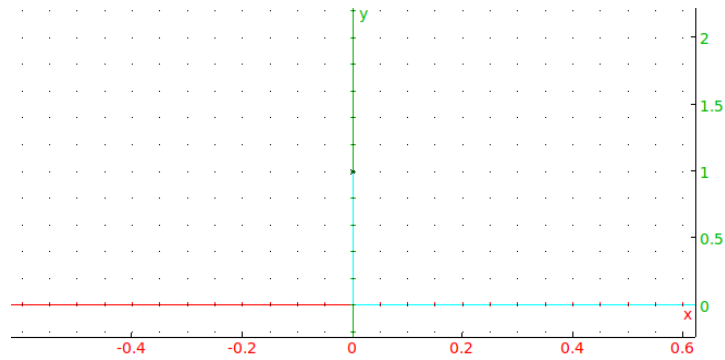


which is the point $(-1,0)$.

Input:

```
single_inter(circle(0,1),line(-1,i),[-1])
```

Output:



which is the point $(0, 1)$. Similarly, since this second point of intersection is closest to $(0, 1/2)$, entering:

Input:

```
single_inter(circle(0,1),line(-1,i),i/2)
```

also draws the second point.

13.6.7 Finding the points of intersection of two geometric objects in the plane: `inter`

See Section 14.4.4 p.990 for points of intersection of objects in space.

The `inter` command finds the intersection of two geometric objects in the plane.

- `inter` takes two mandatory arguments and one optional argument.

- * obj_1, obj_2 , two geometric objects.

- * Optionally, P , a point.

`inter(obj1,obj2⟨ P ⟩)` returns a list of points of intersection of obj_1 and obj_2 .

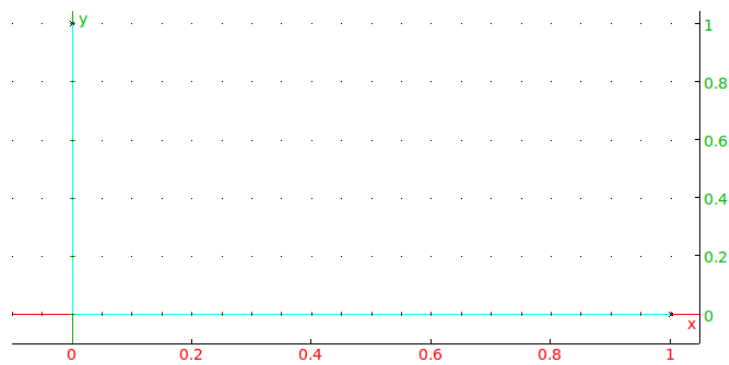
With the argument P , the command returns the point of intersection closest to P .

Examples.

- *Input:*

```
inter(circle(0,1),line(1,i))
```

Output:

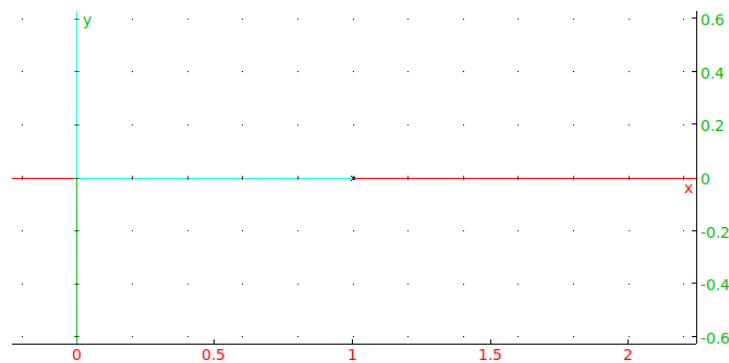


which are the points at $(1,0)$ and $(0,1)$. To get just one of the points, use the usual list indices.

Input:

```
inter(circle(0,1),line(1,i))[0]
```

Output:

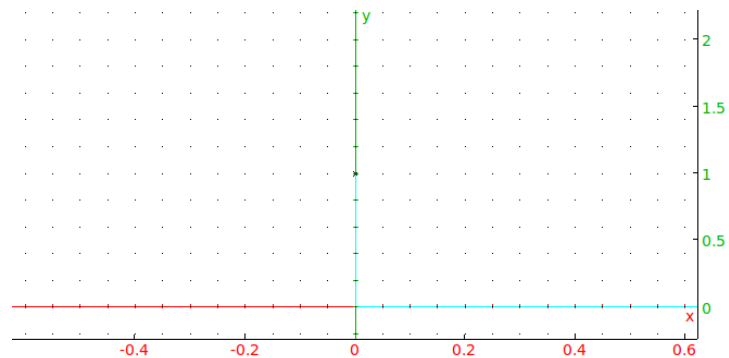


just one of the points. To get the point closest to $(0, 1/2)$:

Input:

```
inter(circle(0,1),line(1,i),i/2)
```

Output:



13.6.8 Finding the orthocenter of a triangle in the plane: orthocenter

The `orthocenter` command finds the orthocenter of a triangle.

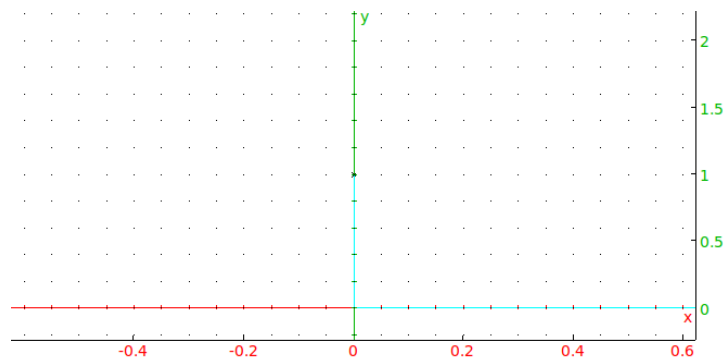
- `orthocenter` takes one argument:
 T , a triangle. The triangle can also be specified with three points.
- `orthocenter(T)` returns the orthocenter of T .

Example.*Input:*

```
orthocenter(triangle(0,1+i,-1+i))
```

or:

```
orthocenter(0,1+i,-1+i)
```

Output:

which is the point $(0,0)$, the orthocenter of the triangle.

13.6.9 Finding the midpoint of a segment in the plane: `midpoint`

See Section 14.4.5 p.992 for midpoints in space.

The `midpoint` command finds the midpoint of two points.

- `midpoint` takes two arguments:
 P, Q , two points (which can also be given as a list).
- `midpoint(P, Q)` draws and returns the midpoint of the segment determined by these points.

13.6.10 The barycenter in the plane: `barycenter`

See Section 14.4.6 p.992 for barycenters of objects in space.

The `barycenter` command returns and draws the barycenter of a set of weighted points.

- **barycenter** takes an unspecified number of arguments:
 L_1, L_2, \dots, L_n , a sequence of lists of length two, where each list consists of a point and a weight. This information can also be given as a matrix with two columns (the first column the points and the second column the weights) or a matrix with two rows and more than two columns.
- **barycenter**(L_1, L_2, \dots, L_n) draws and returns the barycenter of the weighted points.

Example.

The following commands will draw the barycenter of the points (1, 1) with weight 1, (1, -1) with weight 1 and (1, 4) with weight 2.

Input:

```
barycenter([1 + i, 1], [1 - i, 1], [1 + 4*i, 2])
```

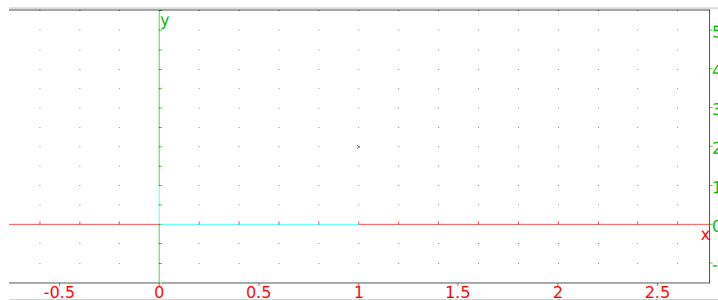
or:

```
barycenter([[1 + i, 1], [1 - i, 1], [1 + 4*i, 2]])
```

or:

```
barycenter([[1 + i, 1 - i, 1 + 4*i], [1, 1, 2]])
```

Output:



13.6.11 The isobarycenter of n points in the plane: **isobarycenter**

See Section 14.4.7 p.993 for isobarycenters of objects in space.

The **isobarycenter** command finds the isobarycenter of a list of points; the isobarycenter is the barycenter when all points are equally weighted.

- **isobarycenter** takes one argument:
 L , a list of points. (The points can also be given by a sequence).
- **isobarycenter**(L) draws and returns the isobarycenter of the points.

13.6.12 The center of a circle in the plane: center

The `center` command finds the center of a circle.

- `center` takes one arguments:
 C , a circle.
- `center(C)` draws and returns the center of C .

Example.

Input:

```
C:= center(circle(point(1+i),1))
```

Output:



13.6.13 The vertices of a polygon in the plane: vertices vertices_abc

The `vertices` command finds the vertices of a polygon.

`vertices_abc` is a synonym for `vertices`.

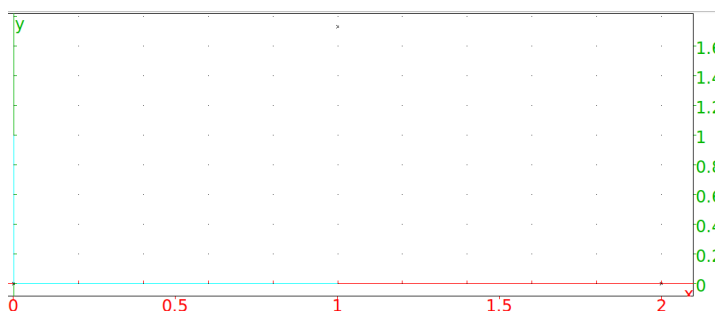
- `vertices` takes one argument:
 P , a polygon.
- `vertices(P)` returns a list of the vertices of P and draws them.

Examples.

– *Input:*

```
vertices(equilateral_triangle(0,2))
```

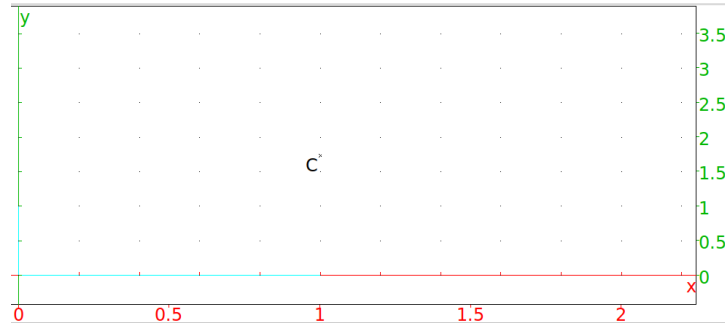
Output:



– *Input:*

```
C:= vertices(equilateral_triangle(0,2))[2]
```

Output:



13.6.14 The vertices of a polygon in the plane, closed: `vertices_abca`

The `vertices_abca` command finds the “closed” list of vertices (it repeats the beginning vertex).

- `vertices_abca` takes one argument:
 P , a polygon.
- `vertices_abca(P)` returns a closed list of the vertices of P and draws them.

13.6.15 A point on a geometric object in the plane: `element`

The `element` command is most useful in a two-dimensional geometry screen; it creates objects that are restricted to a geometric figure.

`element` takes different types of arguments:

- `element` can take one mandatory argument and one optional argument:
 - * $a..b$, a range of values.
 - * Optionally, *init* and *step*, an initial value (by default $(a+b)/2$) and step size (by default $(b-a)/100$).
- `element($a..b$ \langle init,step \rangle)` creates a parameter restricted to the interval from a to b , with the given initial value and whose value can be changed in the given step sizes.

For example, the command `t:= element(0..pi)` creates a parameter t which can take on values between 0 and π and has initial value $\pi/2$. It also creates a slider labeled t which can be used to change the values. The values of any later formulas involving t will change with t .

- **element** can take one mandatory argument and one optional argument:
 - * C , a curve.
 - * Optionally, *init*, an initial value (by default $1/2$).
- **element**($C, \langle init \rangle$) creates a point which will be restricted to the curve, the initial position of the point is determined by setting the parameter (in the default parameterization of the object) to the initial value. If the point can be moved by the mouse (as it can when the geometry screen is in **Pointer** mode), then the motion will be restricted to the geometric object.

For example, the command **A:= element(circle(0,2))** creates a point labeled **A** whose position is restricted to the circle of radius 2 centered at the origin. Since the circle has default parameterization $2 \exp(it)$, **A** starts out at $2 \exp(i/2)$.

- **element** can take two mandatory arguments:
 - * P , a polygon or polygonal line with n sides.
 - * $[\text{floor}(t), \text{frac}(t)]$, where t is a variable previously defined by $t = \text{element}(0..n)$.
- **element**($P, [\text{floor}(t), \text{frac}(t)]$) creates a point restricted to the polygonal line. With the sides numbered starting at 0, the value of $\text{floor}(t)$ determines which side the point is on, and the value of $\text{frac}(t)$ determines how far along the side the point is.

13.7 Lines in plane geometry

13.7.1 Lines and directed lines in the plane: line

See Section 14.5.1 p.993 for lines in space.

The **line** command returns and draws a directed line. It can take its arguments in different ways.

Two points:

- **line** can take two arguments:
 - P, Q , two points (which can also be given as a list).
- **line**(P, Q) returns and draws the line whose direction is from the P to Q .

A point and a slope.

- **line** can take two arguments:
 - * p , a point.
 - * **slope**= m

- `line(p,slope=m)` returns and draws the line through the given point with the given slope, where direction of the line is determined by the slope.

A point and a direction vector.

- `line` can take two arguments:
 - * P , a point.
 - * $[u_1, u_2]$, a direction vector.
- `line($P, [u_1, u_2]$)` returns and draws the line through the given point with the direction given by the direction vector.

An equation.

- `line` can take one argument:
 - $a*x+b*y+c=0$.
- `line($a*x+b*y+c=0$)` returns and draws the line given by the equation. The direction of the line is given by $[b, -a]$.

Example.

Input:

```
line(0,1+i)
```

or:

```
line(1+i,slope=1)
```

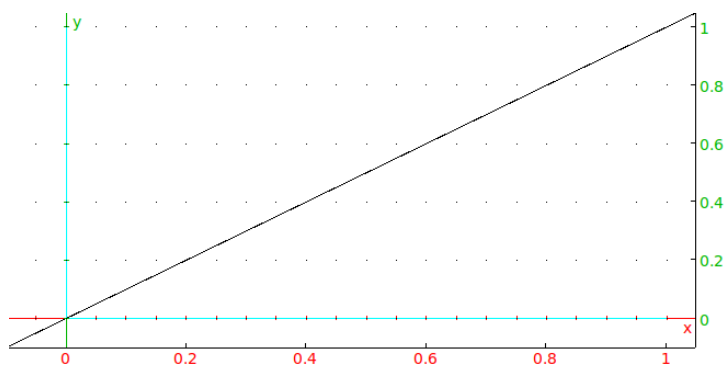
or:

```
line(1+i,[3,3])
```

or:

```
line(y - x = 0)
```

Output:



Warning: To draw a line with an additional argument for color (such as `color=blue`), this argument must be the third argument. In particular, for a list of two points to specify a line in this command, the list must be turned into a sequence, such as with `op`. For example, given a list `L` of two points (possibly the result of a different command) which determines a line, to draw the line `blue` enter `line(op(L),color=blue)`; entering `line(L,color=blue)` will result in an error.

13.7.2 Half-lines in the plane: `half_line`

See Section 14.5.2 p.995 for half-lines in space.

The `half_line` command finds rays.

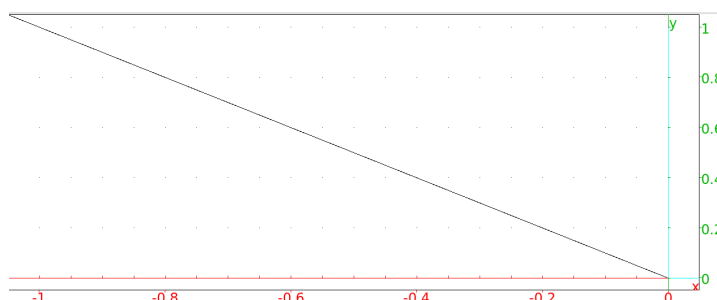
- `half_line` take two arguments:
 P, Q , two points (which can also be given as a list).
- `half_line(P, Q)` returns and draws the ray from P through Q

Example.

Input:

```
half_line(0, -1+i)
```

Output:



13.7.3 Line segments in the plane: `segment` and `Line`

See Section 14.5.3 p.996 for segments in space.

The `segment` command draws line segments. (The `segment` command can also draw vectors (see Section 13.7.4 p.901.)

- `segment` takes two arguments:
 P, Q , two points (which can also be given as a list).
- `segment(P, Q)` returns the corresponding line segment and draws it.

The `Line` command also draws line segments, with a slightly different syntax.

- `Line` takes four arguments:
 a, b, c, d , four real numbers.
- `Line(a, b, c, d)` returns and draws the line segment from (a, b) to (c, d) .

Example.*Input:*

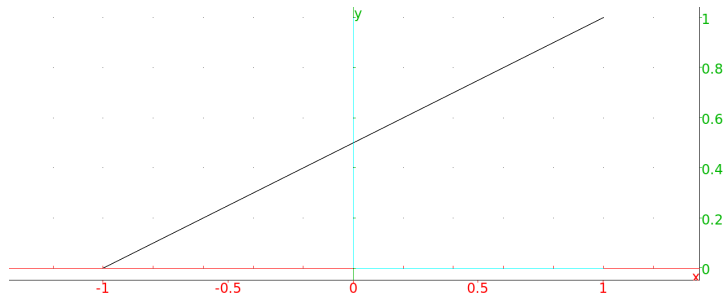
```
segment(-1,1+i)
```

or:

```
segment(point(-1),point(1,1))
```

or:

```
Line(-1,0,1,1)
```

Output:**13.7.4 Vectors in the plane: segment vector**

See Section 14.5.4 p.996 for vectors in space.

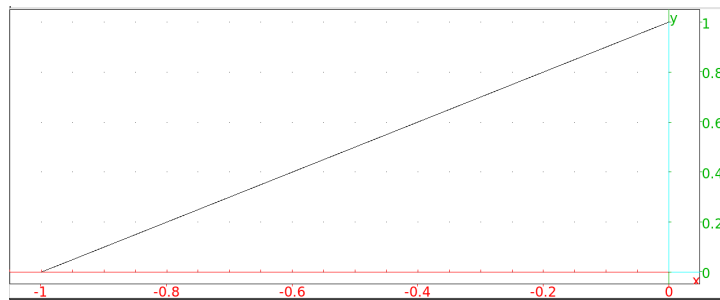
The `segment` commands returns and draws vectors. (The `segment` command can also draw line segments, see section 13.7.3.)

- `segment` takes two arguments:
 - * p , a point.
 - * v , a vector.
- `segment(p, v)` returns the corresponding vector and draws it as a line segment from p to $p + v$.

Example.*Input:*

```
segment([-1,0],[1,1])
```

Output:



The **vector** command also makes vectors, with a different syntax. It can take its arguments in different ways.

The coordinates of the vector.

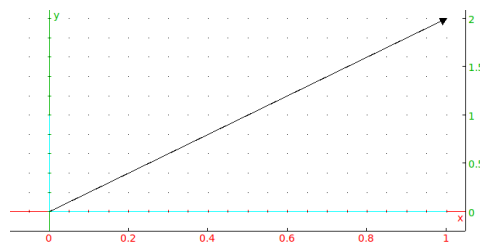
- **vector** takes one argument:
 L , a list of the coordinates of the vector.
- **vector**(L) returns and draws the vector with the given coordinates, starting from the origin.

Example.

Input:

```
vector([1,2])
```

Output:



Two points or a point and a vector.

- **vector** takes two arguments:
 - * P , a point.
 - * Q , a point or a vector. If Q is a point, it can be combined with P in a list.
- **vector**(P, Q) returns and draws the corresponding vector. If the arguments are two points, the vector goes from P to Q . If the arguments are a point and a vector, then the vector starts at P .

Example.

Input:

```
vector([-1,0],[1,i])
```

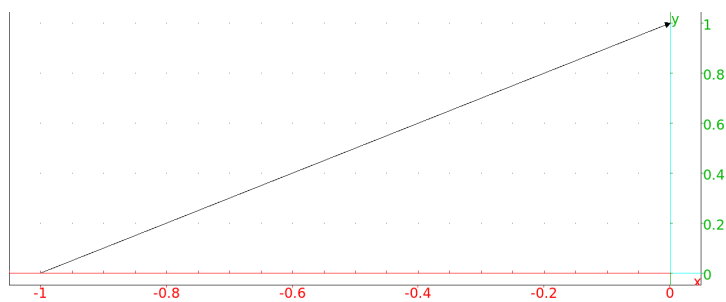
or:

```
vector(-1,i)
```

or:

```
V:= vector(1,2+i):: vector(-1,V)
```

Output:



13.7.5 Parallel lines in the plane: parallel

See Section 14.5.5 p.998 for parallel lines in space.

The `parallel` command finds a line parallel to a given line.

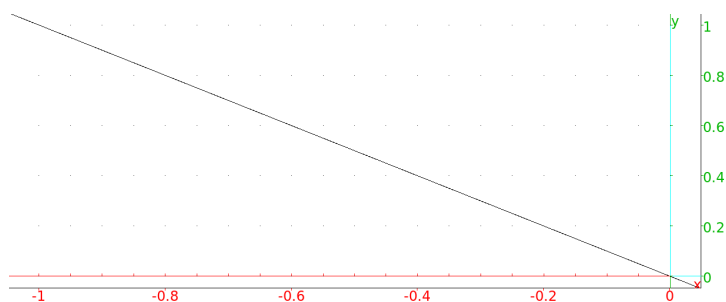
- `parallel` takes two arguments:
 - * p , a point.
 - * ℓ , a line.
- `parallel(p, ℓ)` returns and draws the line parallel to ℓ passing through p .

Example.

Input:

```
parallel(0,line(1,i))
```

Output:



13.7.6 Perpendicular lines in the plane: `perpendicular`

See Section 14.5.6 p.1000 for perpendicular lines in space.

The `perpendicular` command finds a line perpendicular to a given line.

- `perpendicular` takes two arguments:
 - * p , a point.
 - * ℓ , a line. The line can also be specified by giving a sequence of two points on it.
- `perpendicular(p, ℓ)` returns and draws the line perpendicular to ℓ passing through p .

Example.

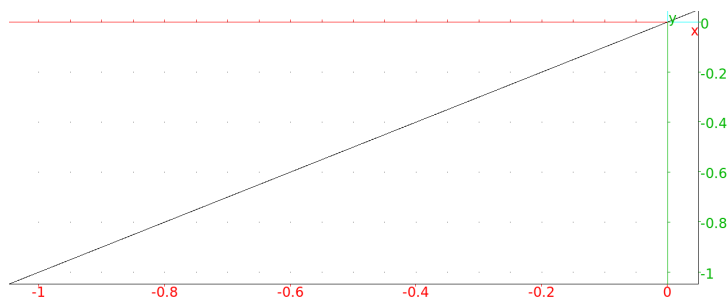
Input:

```
perpendicular(0,line(1,i))
```

or:

```
perpendicular(0,1,i)
```

Output:



13.7.7 Tangents to curves in the plane: `tangent`

See Section 14.6.3 p.1006 for tangents in space.

The `tangent` command finds tangents to curves.

- `tangent` takes one or two arguments:
 - * C , a curve.
 - * p , a point.
- or
 - * e , a point defined with `element` (see Section 13.6.15 p.897) using a curve and parameter value.

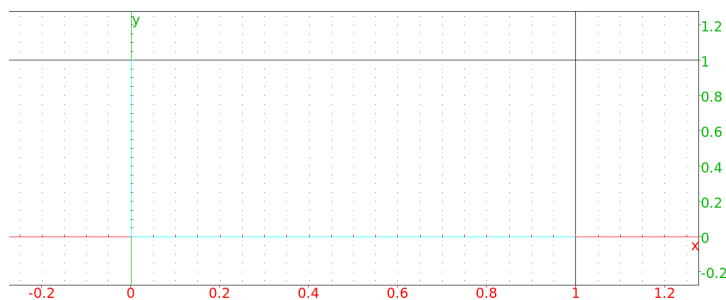
- `tangent(C,p)` (or `tangent(e)`) returns and draws the list of lines tangent to the curve passing through the given point.

Examples.

- *Input:*

```
tangent(circle(0,1),1+i)
```

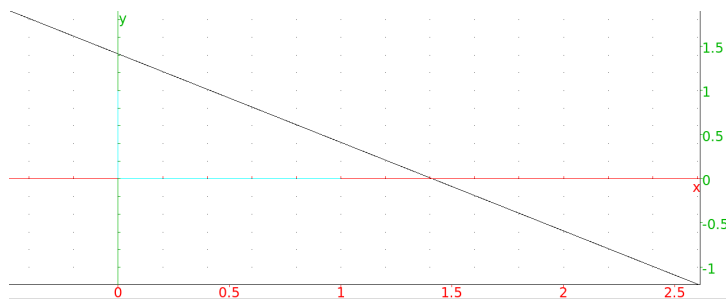
Output:



- *Input:*

```
t:= element(0..pi,pi/4):: A:= element(circle(0,1),t)::  
tangent(A)
```

Output:

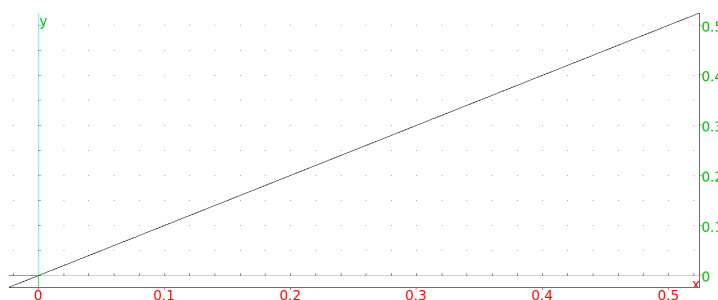


When `tangent` is called with an element, the tangent will change along with the point on the element.

13.7.8 The median of a triangle in the plane: `median_line`

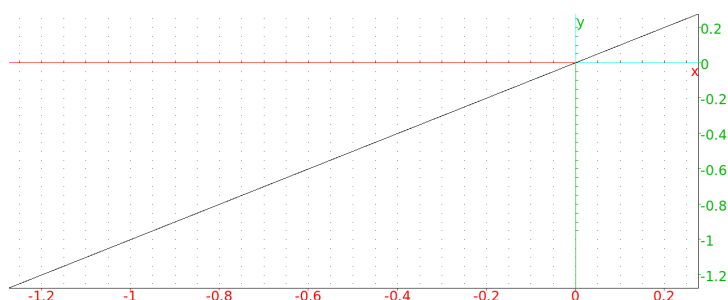
The `median_line` command finds a median line to a triangle.

- `median_line` takes three arguments:
 a, b, c , points.
- `median_line(a,b,c)` returns and draws the median line to the triangle with vertices a, b, c ; through a and bisecting the segment from b to c .

Example.*Input:*`median_line(0,1,i)`*Output:***13.7.9 The altitude of a triangle: altitude**

The `altitude` command finds the altitude line of a triangle.

- `altitude` takes three arguments:
 a, b, c , three points.
- `altitude(a, b, c)` returns and draws the altitude line to the triangle with vertices a, b, c , through a and perpendicular to the segment from b to c .

Example.*Input:*`altitude(0,1,i)`*Output:***13.7.10 The perpendicular bisector of a segment in the plane: perpen_bisector**

See Section 14.6.2 p.1005 for perpendicular bisectors in space.

The `perpen_bisector` command finds the perpendicular bisector of a line segment.

- `perpen_bisector` takes one argument:
 seg , a line segment (or the end points of the segment).

- `perpen_bisector(seg)` returns and draws the perpendicular bisector of seg .

Example.

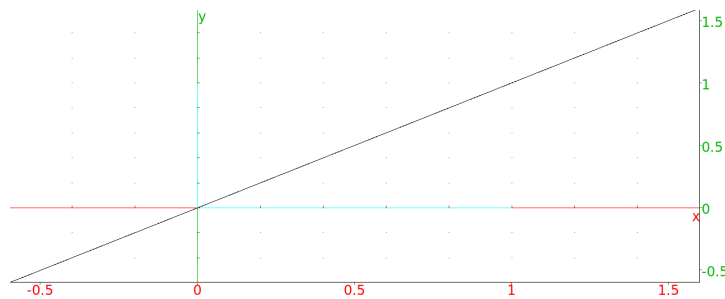
Input:

```
perpen_bisector(1,i)
```

or:

```
perpen_bisector(segment(1,i))
```

Output:



The `perpen_bisector` command can also take two lines as segments, in which case it returns and draws the perpendicular bisector of the segment from the first point defining the first line and the second point defining the second line.

13.7.11 The angle bisector: `bisector`

The `bisector` command finds angle bisectors.

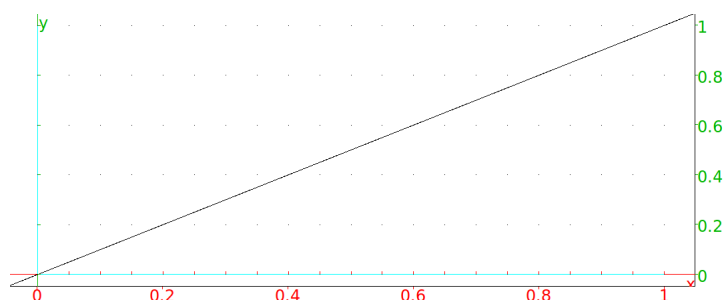
- `bisector` takes three arguments:
 a, b, c , three points (which can also be given as a list).
- `bisector(a, b, c)` returns and draws the bisector of $\angle bac$.

Example.

Input:

```
bisector(0,1,i)
```

Output:



13.7.12 The exterior angle bisector: `exbisector`

The `exbisector` command finds exterior angle bisectors.

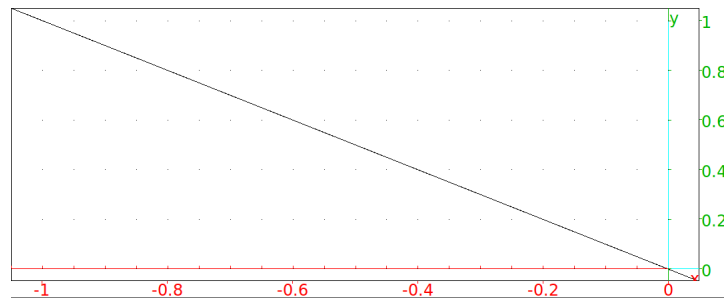
- `exbisector` takes three arguments:
 a, b, c , three points (which can also be given as a list).
- `exbisector(a, b, c)` returns and draws the bisector of the exterior angle of the triangle determined by a, b and c ; a is the vertex of the angle, the opposite of the ray through a and b determine one side of the angle and a and c determine the second side.

Example.

Input:

```
exbisector(0,1,i)
```

Output:



13.8 Triangles in the plane

See Section 14.7 p.1007 for triangles in space.

13.8.1 Arbitrary triangles in the plane: `triangle`

See Section 14.7.1 p.1007 for the `triangle` command in space.

The `triangle` command creates triangles.

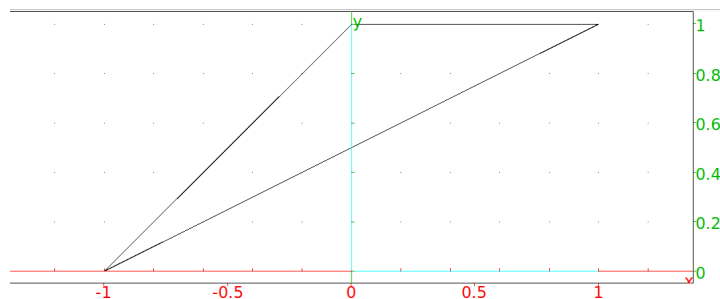
- `triangle` takes three arguments:
 a, b, c , three points (which can be given as a list).
- `triangle(a, b, c)` returns and draws the triangle with vertices a, b and c .

Example.

Input:

```
triangle(-1,i,1+i)
```

Output:



13.8.2 Isosceles triangles in the plane: `isosceles_triangle`

See Section 14.7.2 p.1008 for isosceles triangles in space.

The `isosceles_triangle` command creates isosceles triangles.

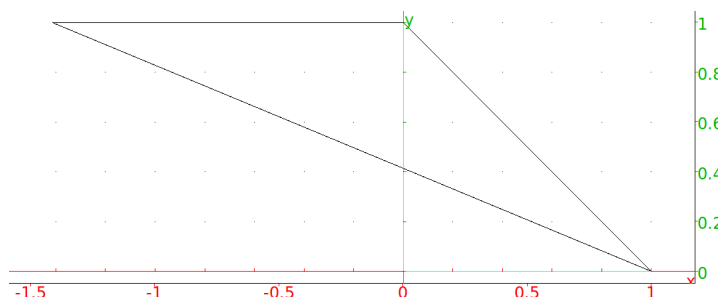
- `isosceles_triangle` takes three mandatory arguments and one optional argument:
 - a, b , two points.
 - θ , an angle.
 - Optionally, `var`, a variable name.
- `isosceles_triangle(a, b, θ (var))` returns and draws the isosceles triangle abc , where ab and ac are equal sides and θ is the angle between ab and ac .
With the argument `var`, c will be assigned to `var`.

Examples.

- *Input:*

```
isosceles_triangle(i, 1, -3*pi/4)
```

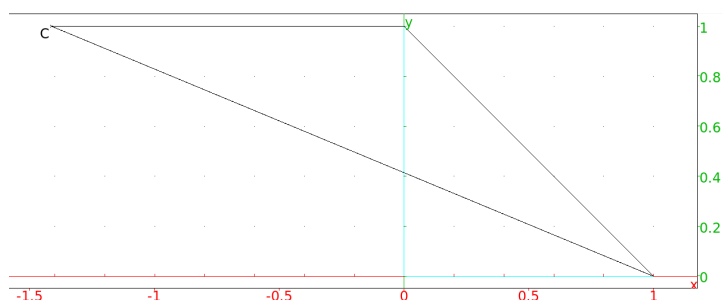
Output:



- *Input:*

```
isosceles_triangle(i, 1, -3*pi/4, C)
```

Output:



Input:

```
normal(affix(C))
```

Output:

$$-\sqrt{2} + i$$

13.8.3 Right triangles in the plane: right_triangle

See Section 14.7.3 p.1010 for right triangles in space.

The `right_triangle` command creates right triangles.

- `right_triangle` takes three mandatory arguments and one optional argument:
 - A, B , two points.
 - k , a nonzero real number.
 - Optionally var , a variable name.
- `right_triangle(A, B, k [var])` returns and draws the right triangle ABC , with the right angle at A and with $\text{length}(AC) = |k| \cdot \text{length}(AB)$. If $k > 0$, then AB to AC is counterclockwise; if $k < 0$ then AB to AC is clockwise.

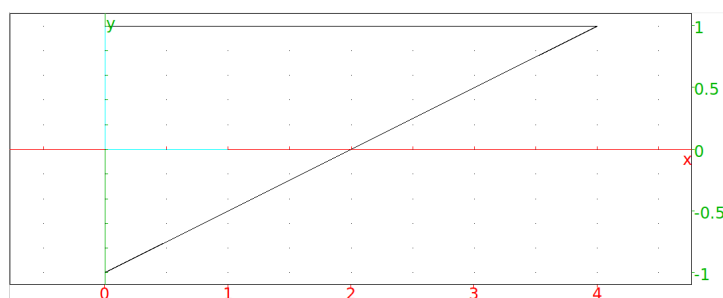
With the argument var , c will be assigned to var .

Examples.

- *Input:*

```
right_triangle(i, -i, 2)
```

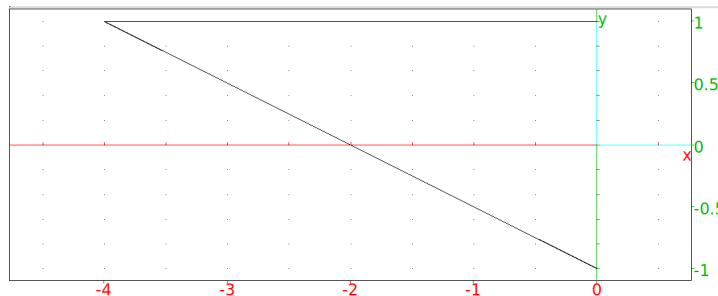
Output:



- *Input:*

```
right_triangle(i, -i, -2)
```

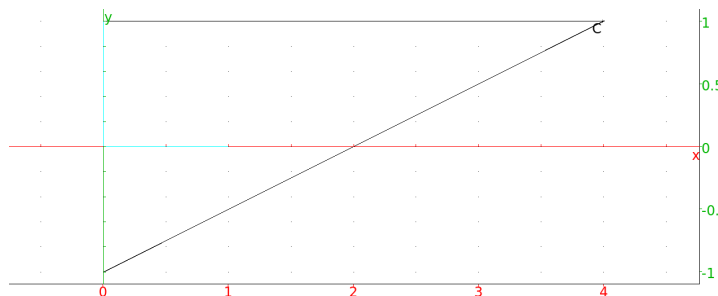
Output:



- *Input:*

```
right_triangle(i, -i, 2, C)
```

Output:



Input:

```
affix(C)
```

Output:

$$4 + i$$

13.8.4 Equilateral triangles in the plane: `equilateral_triangle`

See Section 14.7.4 p.1012 for equilateral triangles in space.

The `equilateral_triangle` command creates equilateral triangles.

- `equilateral_triangle` takes two mandatory arguments and one optional argument:
 - A, B , two points.
 - Optionally, var , a variable name.

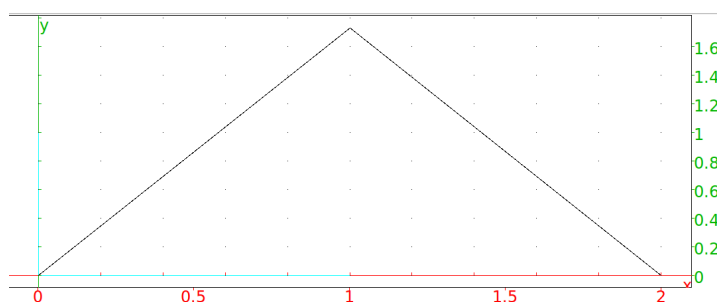
- `equilateral_triangle(A, B <var>)` returns and draws the equilateral ABC , where AB to AC is counterclockwise. With the argument var , C will be assigned to var .

Examples.

- *Input:*

```
equilateral_triangle(0,2)
```

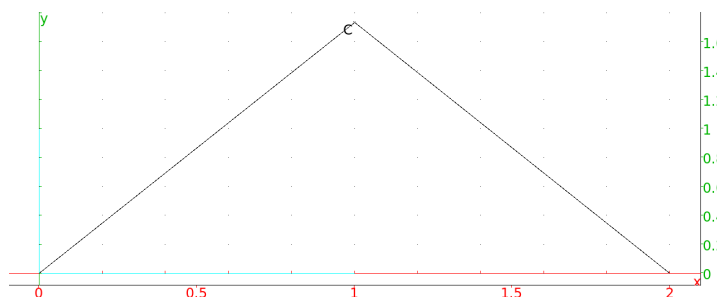
Output:



- *Input:*

```
equilateral_triangle(0, 2, C)
```

Output:



Input:

```
affix(C)
```

Output:

$$(\sqrt{3}i + 1)$$

13.9 Quadrilaterals in the plane

See Section 14.8 p.1013 for quadrilaterals in space.

13.9.1 Squares in the plane: square

See Section 14.8.1 p.1013 for squares in space.

The `square` command creates squares.

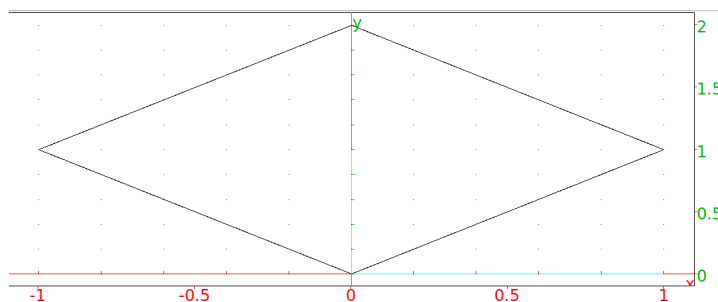
- `square` takes two mandatory arguments and two optional arguments.
 - A, B , two points.
 - Optionally, $varc, vard$, two variable names.
- `square($A, B \langle varc, vard \rangle$)` returns and draws the square $ABCD$, where the square is traversed counterclockwise.
If the arguments $varc$ and $vard$ are given, then C and D will be assigned to them.

Examples.

- *Input:*

```
square(0,1+i)
```

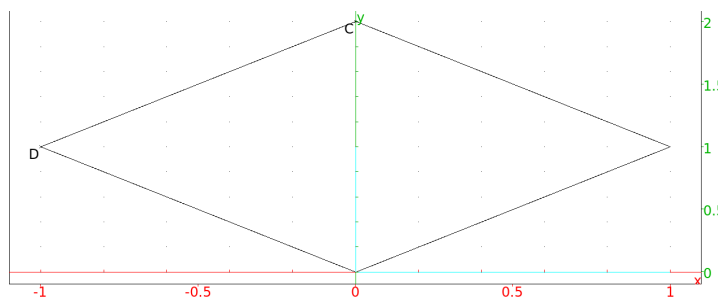
Output:



- *Input:*

```
square(0,1+i,C,D)
```

Output:



Input:

```
affix(C), affix(D)
```

Output:

$2i, -1 + i$

13.9.2 Rhombuses in the plane: rhombus

See Section 14.8.2 p.1014 for rhombuses in space.

The `rhombus` command creates rhombuses.

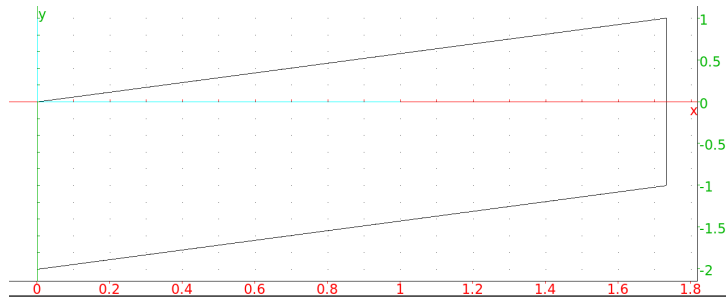
- `rhombus` takes three mandatory arguments and two optional arguments.
 - A, B , two points.
 - a , a real number.
 - Optionally, $varc$, $vard$, two variable names.
- `rhombus(A, B, a ($varc, vard$))` returns and draws the rhombus $ABCD$, where a is the counterclockwise angle from AB to AC . If the arguments $varc$ and $vard$ are given, then C and D will be assigned to them.

Examples.

- *Input:*

```
rhombus(-2*i, sqrt(3) - i, pi/3)
```

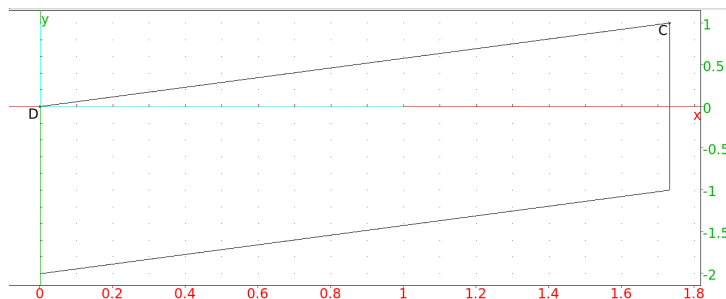
Output:



- *Input:*

```
rhombus(-2*i, sqrt(3) - i, pi/3, C, D)
```

Output:



Input:

```
affix(C), affix(D)
```

Output:

$$\sqrt{3} + i, 0$$

13.9.3 Rectangles in the plane: rectangle

See Section 14.8.3 p.1016 for rectangles in space.

The `rectangle` creates rectangles.

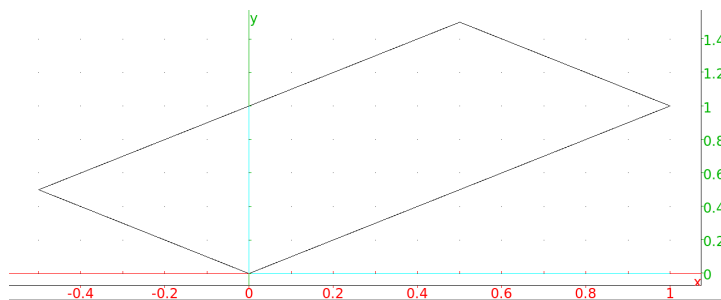
- `rectangle` takes three mandatory arguments and two optional arguments:
 - A, B , two points.
 - k , a nonzero real number.
 - Optionally, $varc$, $vard$, two variable names.
- `rectangle(A, B, k ($varc, vard$))` returns and draws the rectangle $ABCD$, where $AD = |k| \cdot AB$ and the angle from AB to AD is counterclockwise if $k > 0$, clockwise if $k < 0$.
If the arguments $varc$ and $vard$ are given, then C and D will be assigned to them.

Examples.

- *Input:*

```
rectangle(0, 1+i, 1/2)
```

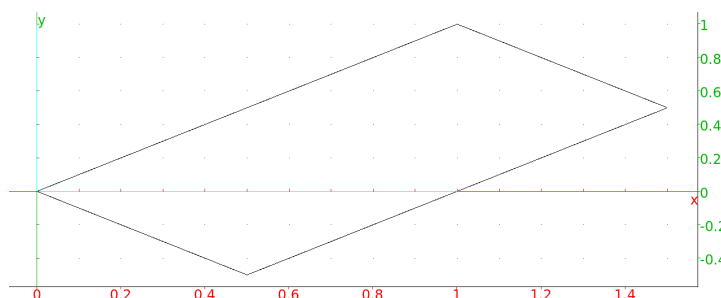
Output:



- *Input:*

```
rectangle(0, 1+i, -1/2)
```

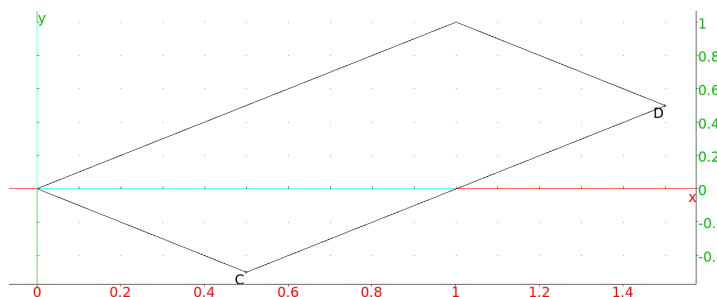
Output:



- *Input:*

```
rectangle(0, 1+i, -1/2, C, D)
```

Output:



Input:

```
affix(C), affix(D)
```

Output:

$$\frac{3+i}{2}, \frac{1-i}{2}$$

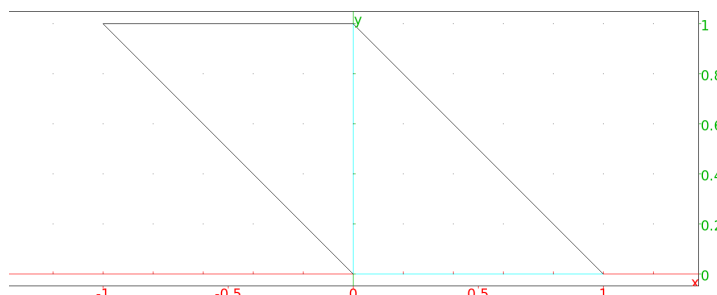
Given `rectangle(A, B, k)`, Xcas computes D by $\text{affix}(D) = \text{affix}(A) + k \exp(i\pi/2)(\text{affix}(B) - \text{affix}(A))$. If k is complex, then `rectangle` draws a parallelogram.

Example.

Input:

```
rectangle(0,1,1+i)
```

Output:



13.9.4 Parallelograms in the plane: parallelogram

See Section 14.8.4 p.1018 for parallelograms in space.

The `parallelogram` command creates parallelogram.

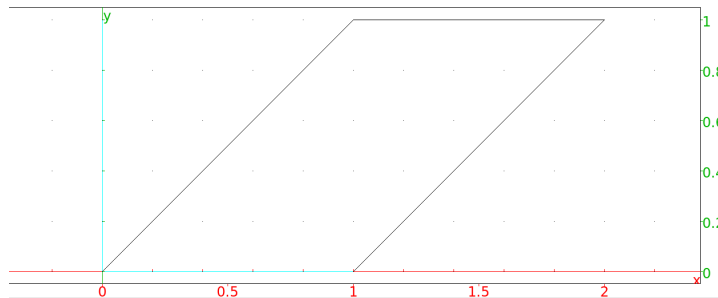
- `parallelogram` takes three mandatory arguments and one optional argument:
 - A, B, C , three points.

- Optionally, *var*, a variable name.
- `parallelogram(A, B, C (var)`) returns and draws the parallelogram $ABCD$ for the appropriate D .
If the argument *var* is given, then D will be assigned to it.

Examples.

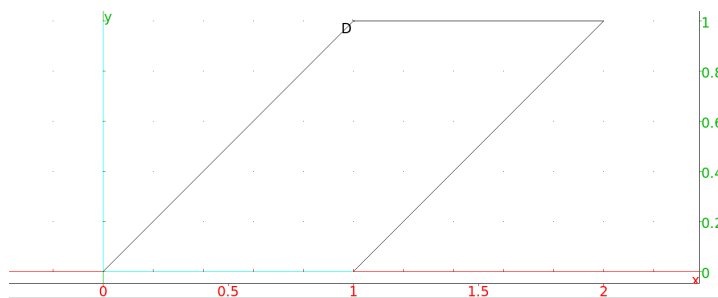
- *Input:*

```
parallelogram(0, 1, 2 + i)
```

Output:

- *Input:*

```
parallelogram(0, 1, 2 + i, D)
```

Output:*Input:*

```
affix(D)
```

Output:
 $1 + i$

13.9.5 Arbitrary quadrilaterals in the plane: `quadrilateral`

See Section 14.8.5 p.1019 for quadrilaterals in space.

The `quadrilateral` creates arbitrary quadrilaterals.

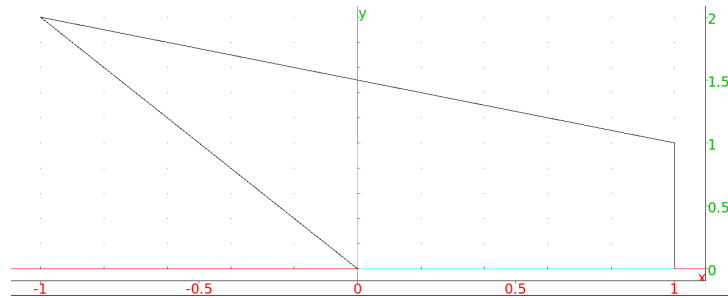
- `quadrilateral` takes four arguments:
 A, B, C, D , four points.
- `quadrilateral(A, B, C, D)` returns and draws the quadrilateral $ABCD$.

Example.

Input:

```
quadrilateral(0, 1, 1 + i, -1 + 2*i)
```

Output:



13.10 Other polygons in the plane

See Section 14.9 p.1019 for polygons in space.

13.10.1 Regular hexagons in the plane: `hexagon`

See Section 14.9.1 p.1020 for hexagons in space.

The `hexagon` command creates hexagons.

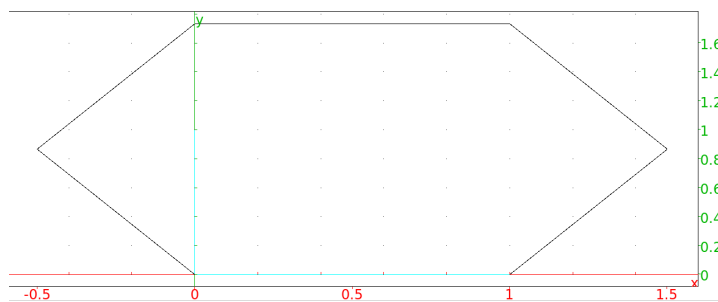
- `hexagon` takes two mandatory arguments and four optional arguments:
 - A, B , two points.
 - Optionally, `varc`, `vard`, `vare`, `varf`, variable names.
- `hexagon(A, B, C <varc, vard, vare>)` returns and draws the regular hexagon $ABCDEF$, where the vertices are counterclockwise.
 If the arguments `varc`, `vard`, `vare`, `varf` are given, then the points C, D, E and F will be assigned to them.

Examples.

- *Input:*

```
hexagon(0, 1)
```

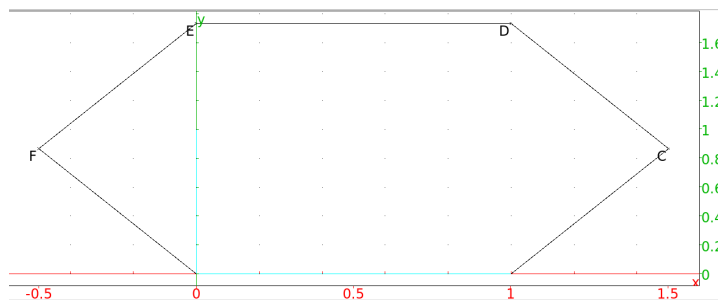
Output:



- *Input:*

`hexagon(0, 1, C, D, E, F)`

Output:



Input:

`affix(C), affix(D), affix(E), affix(F)`

Output:

$$\frac{\sqrt{3}i + 1}{2} + 1, \frac{2}{2}(\sqrt{3}i + 1), \frac{2}{2}(\sqrt{3}i + 1) - 1, \frac{\sqrt{3}i + 1}{2} - 1$$

13.10.2 Regular polygons in the plane: isopolygon

See Section 14.9.2 p.1020 for regular polygons in space.

The `isopolygon` command creates regular polygons.

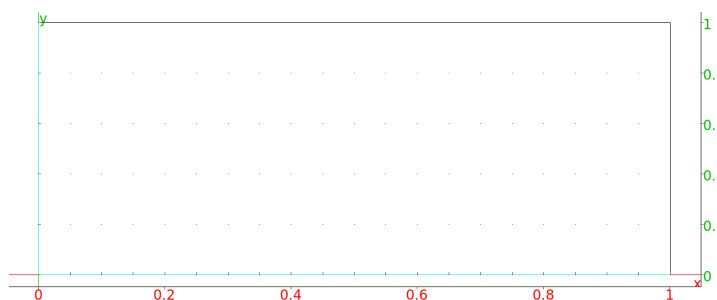
- `isopolygon` takes three arguments:
 - A, B , two points.
 - k , a nonzero integer.
- `isopolygon(A, B, k)` returns and draws the regular $|k|$ -sided polygon with one side AB . If $k > 0$, then the polygon will continue counter-clockwise; if $k < 0$, then it will be clockwise.

Examples.

- *Input:*

```
isopolygon(0,1,4)
```

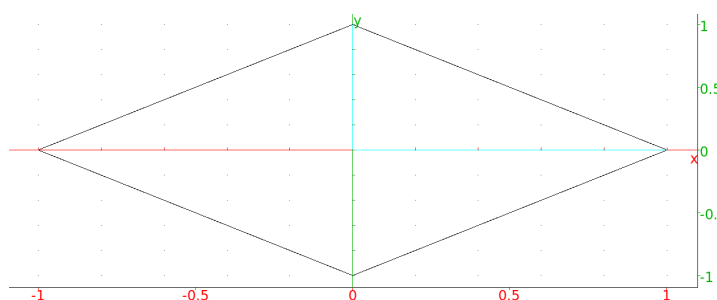
Output:



Input:

```
isopolygon(0,1,-4)
```

Output:



13.10.3 General polygons in the plane: `polygon`

See Section 14.9.3 p.1022 for general polygons in space.

The `polygon` command draws general polygons.

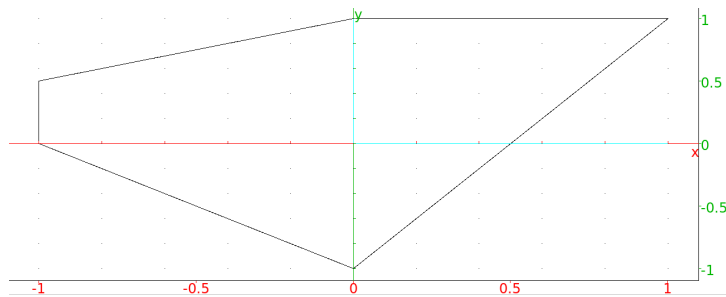
- `polygon` takes an unspecified number of arguments:
points, a sequence or list of points.
- `polygon(points)` returns and draws the polygon with vertices given by the points.

Examples.

- *Input:*

```
polygon(-1,-1+i/2,i,1+i,-i)
```

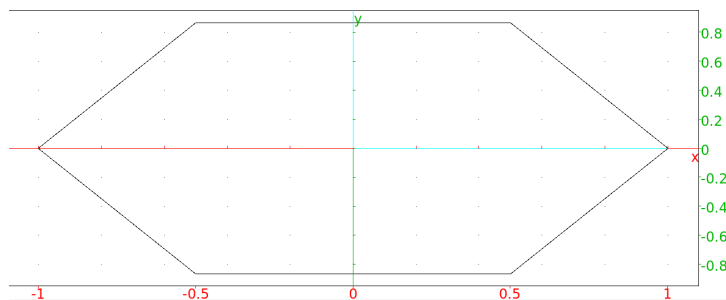
Output:



- *Input:*

```
polygon(makelist(x->exp(i*pi*x/3),0,5,1))
```

Output:



13.10.4 Polygonal lines in the plane: open_polygon

See Section 14.9.4 p.1023 for polygonal lines in space.

The `open_polygon` command draws a polygonal path.

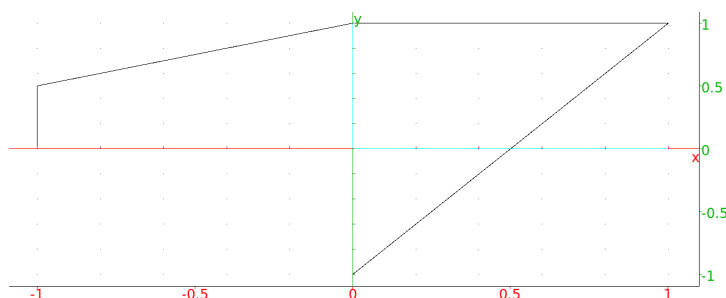
- `open_polygon` takes an unspecified number of points: *points*, a sequence or list of points.
- `open_polygon(points)` returns and draws the polygon line with the vertices given by the points.

Examples.

- *Input:*

```
open_polygon(-1,-1+i/2,i,1+i,-i)
```

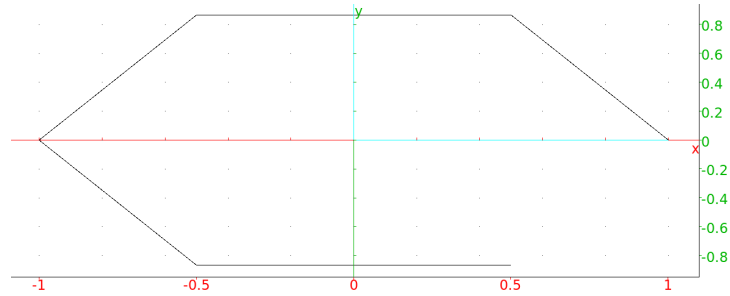
Output:



- *Input:*

```
open_polygon(makelist(x->exp(i*pi*x/3),0,5,1))
```

Output:



13.10.5 Convex hulls: convexhull

The `convexhull` command uses the Graham scanning algorithm to find the convex hull of a set of points.

- `convexhull` takes an unspecified number of arguments:
points, a sequence or list of points.
- `convexhull(points)` returns the vertices of the convex hull of the points.

Example.

Input:

```
convexhull(0,1,1+i,1+2i,-1-i,1-3i,-2+i)
```

Output:

```
1 - 3i, 1 + 2i, -2 + i, -1 - i
```

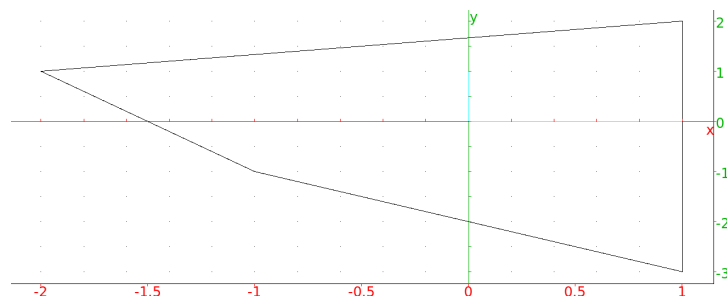
To draw the hull, you can use the `polygon` command with the output of `convexhull` (see Section 13.10.3 p.920).

Example.

Input:

```
polygon(convexhull(0,1,1+i,1+2i,-1-i,1-3i,-2+i))
```

Output:



13.11 Circles

13.11.1 Circles and arcs in the plane: circle

See also Section 13.11.2 p.925.

See Section 14.10 p.1023 for circles in space.

The `circle` command creates circles and arcs. You can specify the circle in different ways.

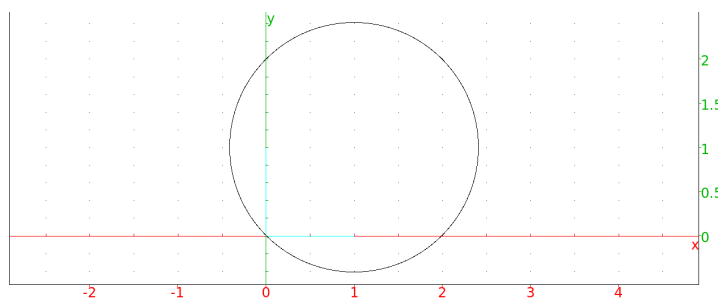
- – `circle` can take one argument:
 - eqn*, the equation of a circle with variables x and y (or an expression assumed to be set to 0).
- `circle(eqn)` returns and draws the circle.

Example.

Input:

```
circle(x^2 + y^2 - 2*x - 2*y)
```

Output:



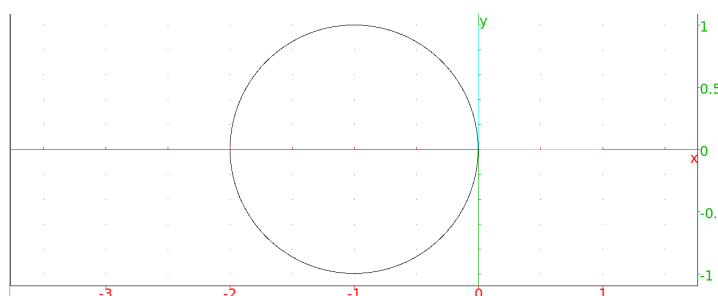
- – `circle` can take two arguments:
 - * P , a point.
 - * α , a complex number.
- `circle(P, α)` returns and draws the circle centered at P and whose radius is $|\alpha|$.

Example.

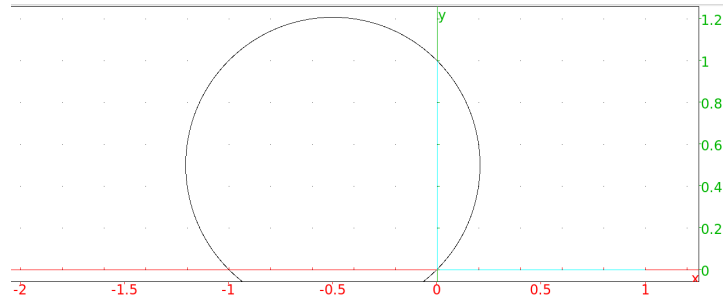
Input:

```
circle(-1,i)
```

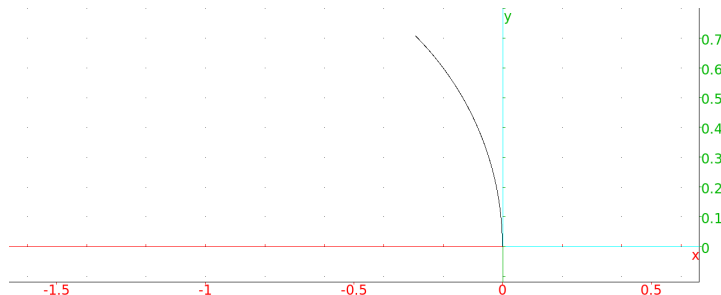
Output:



- – `circle` can take two arguments:
 A, B , two points (where B must be the value of `point` and not simply the affix).
 - `circle(A, B)` returns and draws the circle whose diameter is AB .

Example.*Input:*`circle(-1,point(i))`*Output:*

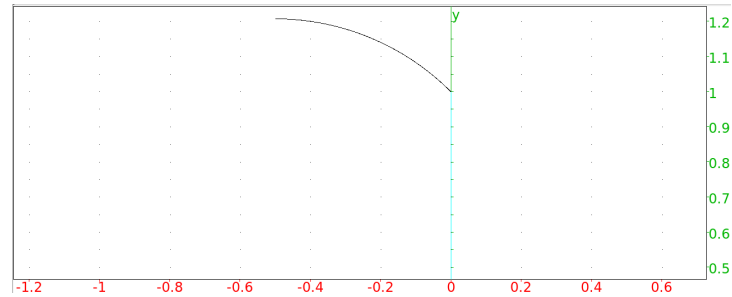
- – `circle` can take four mandatory arguments and two optional arguments:
 - * C , a point.
 - * r , a complex number.
 - * a, b , two real numbers.
 - * Optionally, var_1, var_2 , variable names.
- `circle(C, r, a, b)` returns and draws an arc of the circle with center C and radius $|r|$, with central angles a and b . The angles start on the axis defined by C and $C + r$.
 If the arguments var_1 and var_2 are given, they will be assigned to the ends of the arc.

Examples.– *Input:*`circle(-1,1,0,pi/4)`*Output:*

– *Input:*

```
circle(-1,point(i),0,pi/4)
```

Output:



13.11.2 Circular arcs: arc

See also Section 13.11.1 p.923

The `arc` command creates circular arcs.

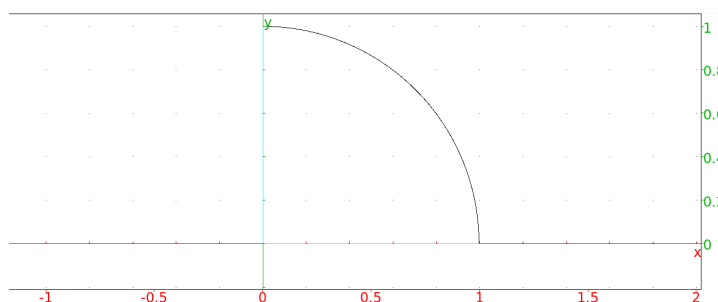
- `arc` takes three mandatory arguments and two optional arguments:
 - A, B , two points.
 - a , a real number between -2π and 2π .
 - Optionally, `varc`, `varr`, two variable names.
- `arc(A, B, a ⟨⟩)` returns and draws the circular arc from A to B that represents an angle of a . (Note that the center of the circle will be $(A + B)/2 + i * (B - A)/(2 \tan(a/2))$.)
If the arguments `varc`, `varr` are given, they will be assigned the center and radius of the circle.

Examples.

- *Input:*

```
arc(1,i,pi/2)
```

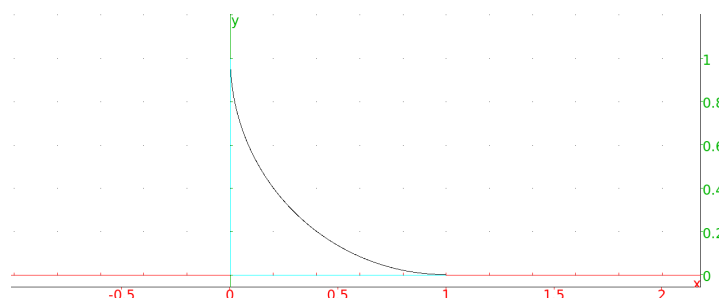
Output:



- *Input:*

```
arc(1,i,-pi/2)
```

Output:



13.11.3 Circles (TI compatibility): Circle

The `Circle` command creates a circle.

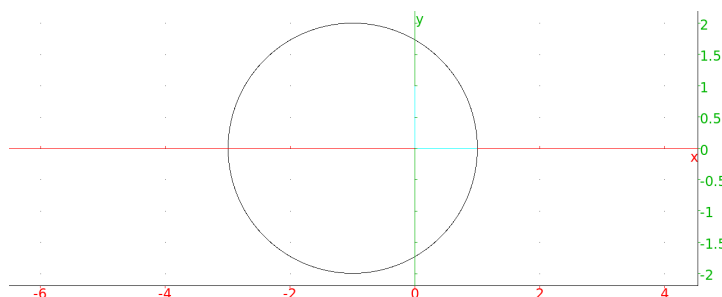
- `Circle` takes three mandatory arguments and one optional argument:
 - x, y, r , three real numbers.
 - Optionally, n , either 0 or 1 (by default, 1).
- `Circle(x, y, r, n)` returns the circle centered at (x, y) with radius r . If $n = 1$, it also draws the circle; if $n = 0$, it erases it.

Example.

Input:

```
Circle(-1,0,2)
```

Output:



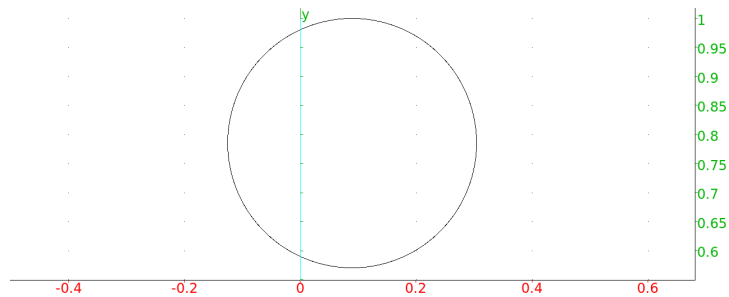
13.11.4 Inscribed circles: incircle

The `incircle` command creates the inscribed circle of a triangle.

- `incircle` takes three arguments:
 - A, B, C , three points.
- `incircle(A, B, C)` returns and draws the circle inscribed in triangle ABC .

Example.*Input:*

```
incircle(-1,i,1+i)
```

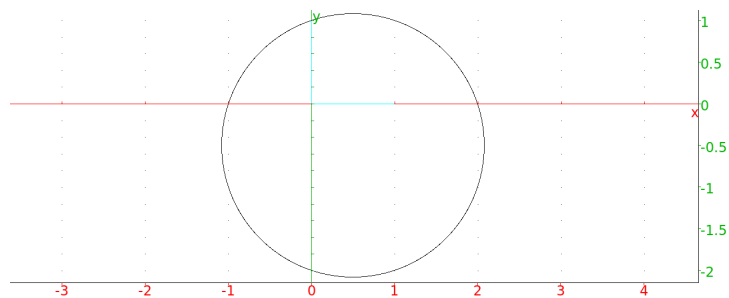
Output:**13.11.5 Circumscribed circles: circumcircle**

The `circumcircle` command creates the circumscribed circle of a triangle.

- *circumcircle* takes three arguments:
 A, B, C , three points.
- `circumcircle(A, B, C)` returns and draws the circle circumscribed about triangle ABC .

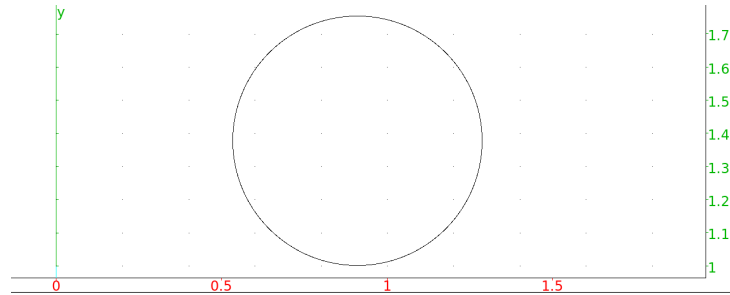
Example.*Input:*

```
circumcircle(-1,i,1+i)
```

Output:**13.11.6 Excircles: excircle**

The `excircle` draws an excircle of a triangle.

- *excircle* takes three arguments:
 A, B, C , three points.
- `excircle(A, B, C)` returns and draws the excircle of the triangle ABC in the interior angle of A .

Example.*Input:*`excircle(-1,i,1+i)`*Output:***13.11.7 The power of a point relative to a circle: powerpc**

Given a circle C of radius r and a point A at a distance of d from the center of C , the power of A relative to C is $d^2 - r^2$.

The `powerpc` command finds the power of a point relative to a circle.

- `powerpc` takes two arguments:
 - C , a circle.
 - P , a point.
- `powerpc(C,P)` returns the power of P relative to C .

Example.*Input:*`powerpc(circle(0, 1+i), 3+i)`*Output:*

8

13.11.8 The radical axis of two circles: radical_axis

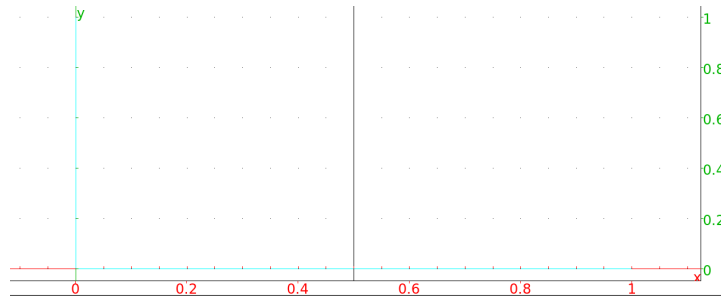
The radical axis of two circles is the set of points which have the same power with respect to each circle.

The `radical_axis` command finds the radical axis of two circles.

- `radical_axis` takes two arguments:
 - C_1, C_2 , two circles.
- `radical_axis(C1, C2)` returns and draws the radical axis of C_1 and C_2 .

Example.*Input:*`radical_axis(circle(0,1+i),circle(1,1+i))`

Output:



13.12 Other conic sections

13.12.1 The ellipse in the plane: ellipse

See Section 14.11.1 p.1025 for ellipses in space.

The `ellipse` command draws ellipses and other conic sections.

`ellipse` can take parameters in different forms.

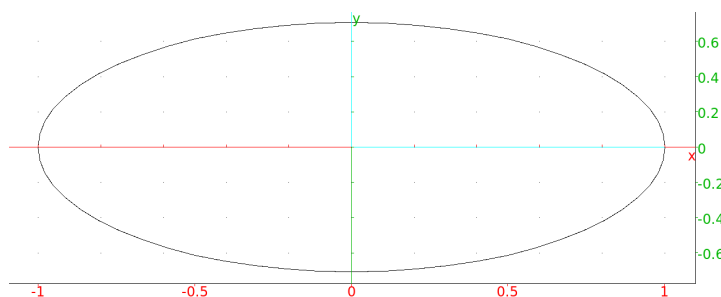
- – `ellipse` can take one parameter:
 eqn , a second degree equation in the variables x and y (or an expression which will be set to zero).
- `ellipse(eqn)` returns and draws the conic section given by eqn .

Example.

Input:

```
ellipse(x^2 + 2*y^2 - 1)
```

Output:



- – `ellipse` can take three arguments:
 - * A, B , two points.
 - * C , a point or a real number.
- `ellipse(A, B, C)` returns and draws the ellipse with foci A and B and passing through C (if C is a point) or whose semi-major axis has length C (if C is a real number).

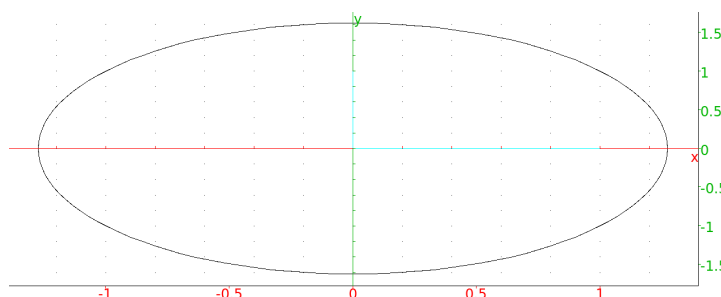
Note that if the third argument is a point on the real axis, the real affix of the point won't work, it needs to be specified with the `point` command.

Examples.

- *Input:*

```
ellipse(-i,i,i+1)
```

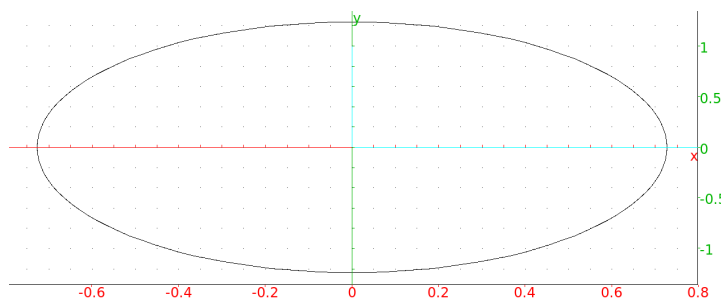
Output:



- *Input:*

```
ellipse(-i,i,sqrt(5) - 1)
```

Output:



13.12.2 The hyperbola in the plane: hyperbola

See Section 14.11.2 p.1025 for hyperbolas in space.

The `hyperbola` command draws hyperbolas and other conic sections. `hyperbola` can take parameters in different forms.

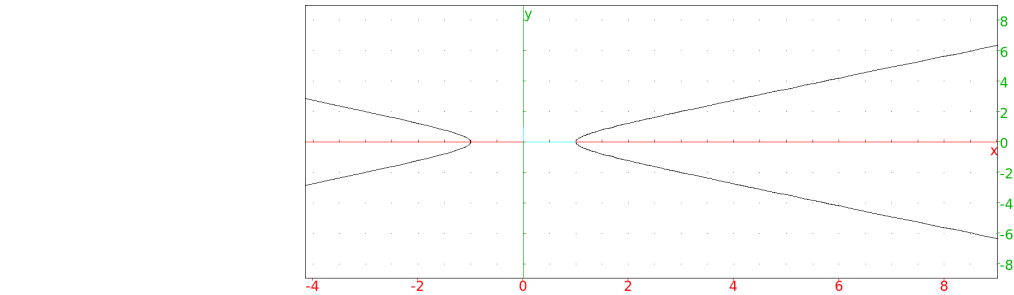
- – `hyperbola` can take one argument:
 eqn , a second degree equation in the variables x and y (or an expression which will be set to zero).
- `hyperbola(eqn)` returns and draws the conic section given by the equation eqn .

Example.

Input:

```
hyperbola(x^2 - 2*y^2 - 1)
```

Output:



- – `hyperbola` can take three arguments:
 - * A, B , two points.
 - * C , a point or a real number.
- `hyperbola(A, B, C)` returns and draws the hyperbola with foci A and B and passing through C (if C is a point) or whose semi-major axis has length C (if C is a real number).

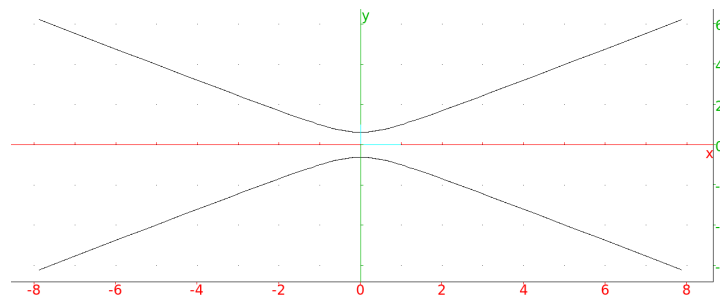
Note that if the third argument is a point on the real axis, the real affix of the point won't work, it needs to be specified with the `point` command.

Examples.

– *Input:*

`hyperbola(-i,i,i+1)`

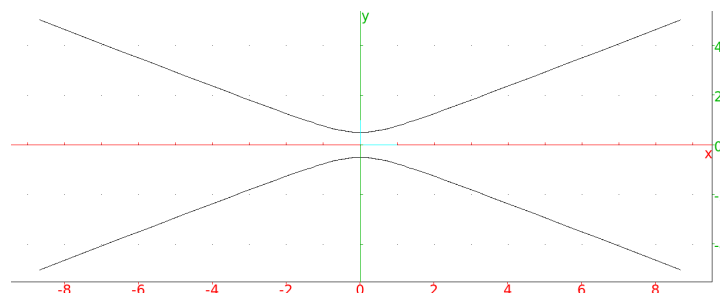
Output:



– *Input:*

`hyperbola(-i,i,1/2)`

Output:



13.12.3 The parabola in the plane: parabola

See Section 14.11.3 p.1026 for parabolas in space.

The `parabola` command draws parabolas and other conic sections. `parabola` can take parameters in different forms.

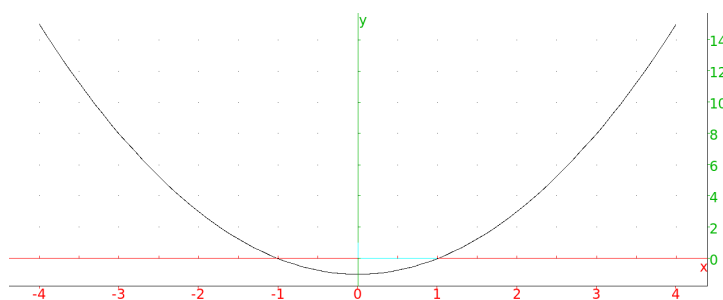
- `parabola` can take one argument:
eqn, a second degree equation in the variables *x* and *y* (or an expression which will be set to zero).
- `parabola(eqn)` returns and draws the conic section given by the equation *eqn*.

Example.

Input:

```
parabola(x^2 - y - 1)
```

Output:



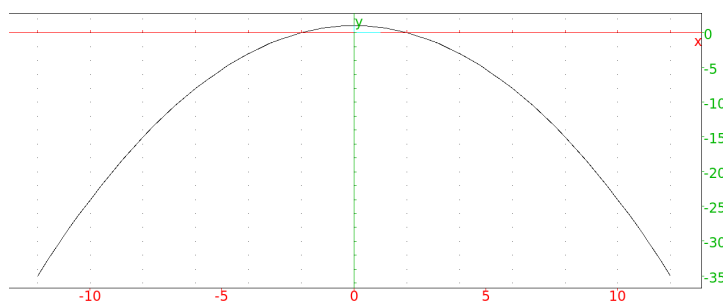
- `parabola` can take two arguments:
F, *V*, two points.
- `parabola(F, V)` returns and draws the parabola with focus *F* and vertex *V*.

Example.

Input:

```
parabola(0,i)
```

Output:



- `parabola` can take two parameters:

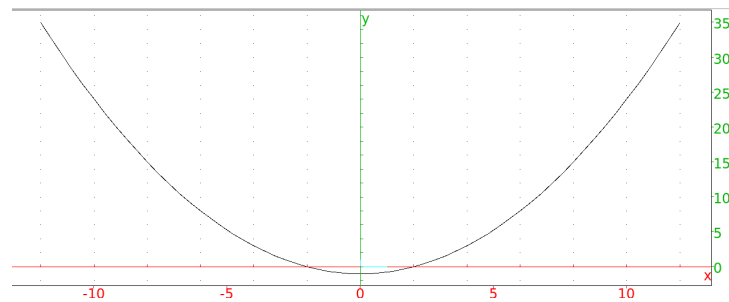
- * $A = (a, b)$, a point.
- * c , a real number.
- `parabola(A, c)` returns and draws the parabola $y = b + c(x - a)^2$.

Examples.

- *Input:*

`parabola(-i, 1)`

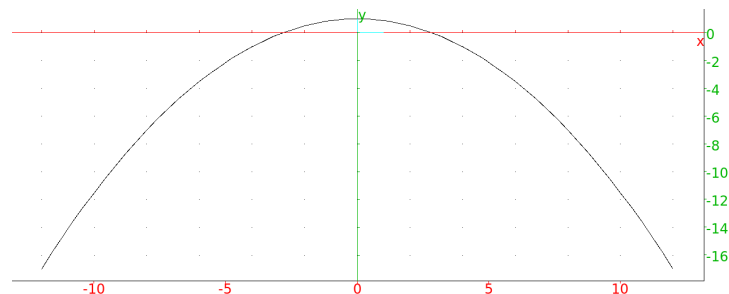
Output:



- *Input:*

`parabola(-i, i, 1/2)`

Output:



13.13 Coordinates in the plane

13.13.1 The affix of a point or vector: `affix`

The `affix` command finds the affix of a point or vector; namely, the complex number corresponding to the point or vector.

- `affix` takes one arguments:
 P , a point or vector.
- `affix(P)` returns the affix of P .

Examples.

- *Input:*

`affix(point(2, 3))`

Output:

$$2 + 3i$$

- *Input:*

```
affix(vector(-1,i))
```

Output:

$$1 + i$$

13.13.2 The abscissa of a point or vector in the plane: `abscissa`

See Section 14.12.1 p.1027 for abscissas in three-dimensional geometry.

The `abscissa` command finds the abscissa (x -coordinate) of a point.

- `abscissa` takes one argument:
 P , a point.
- `abscissa(P)` returns the abscissa of P .

Examples.

- *Input:*

```
abscissa(point(1 + 2*i))
```

Output:

$$1$$

- *Input:*

```
abscissa(point(i) - point(1 + 2*i))
```

Output:

$$-1$$

- *Input:*

```
abscissa(1 + 2*i)
```

Output:

$$1$$

- *Input:*

```
abscissa([1,2])
```

Output:

$$1$$

13.13.3 The ordinate of a point or vector in the plane: ordinate

See Section 14.12.2 p.1027 for ordinates in three-dimensional geometry.

The `ordinate` command finds the ordinate (y coordinate) of a point.

- `ordinate` takes one argument:
 P , a point.
- `ordinate(P)` returns the ordinate of P .

Examples.

- *Input:*

```
ordinate(point(1 + 2*i))
```

Output:

2

- *Input:*

```
ordinate(point(i) - point(1 + 2*i))
```

Output:

-1

- *Input:*

```
ordinate(1 + 2*i)
```

Output:

2

- *Input:*

```
ordinate([1,2])
```

Output:

2

13.13.4 The coordinates of a point, vector or line in the plane: coordinates

See Section 14.12.4 p.1028 for coordinates in three-dimensional geometry.

The `coordinates` finds the coordinates of a point or two points that determine a line.

- `coordinates` takes one argument:
 X , either a point, a sequence or list of points, or a line.
- `coordinates(X)` returns:

- a list consisting of the abscissa and ordinate of X , if X is a point or a vector, or a sequence or list of such lists, if X is a sequence or list of points.
- a list of two points on the line X , in the order determined by the direction of the line, if X is a line.

Examples.

- *Input:*

`coordinates(1+2*i)`

or:

`coordinates(point(1+2*i))`

or:

`coordinates(vector(1+2*i))`

Output:

`[1,2]`

- *Input:*

`coordinates(point(1+2*i) - point(i))`

or:

`coordinates(vector(i,1+2*i))`

or:

`coordinates(vector(point(i),point(1+2*i)))`

or:

`coordinates(vector([0,1],[1,2]))`

Output:

`[1,1]`

- *Input:*

`d:= line(-1+i,1+2*i)`

or:

`d:= line(point(-1,1),point(1,2))`

then:

```
coordinates(d)
```

Output:

$$[-1 + i, 1 + 2i]$$

- *Input:*

```
coordinates(line(y = (1/2 * x + 3/2)))
```

Output:

$$\left[\frac{3i}{2}, 1 + 2i \right]$$

- *Input:*

```
coordinates(line(x - 2*y + 3 = 0))
```

Output:

$$\left[\frac{3i}{2}, \frac{-4 + i}{2} \right]$$

- *Input:*

```
coordinates(i, 1+2*i)
```

or:

```
coordinates(point(i), point(1+2*i))
```

Output:

$$[0, 1], [1, 2]$$

- Note that if the argument is a list of real numbers, it is interpreted as a list of points on the real axis.

Input:

```
coordinates([1, 2])
```

Output:

$$\begin{bmatrix} 1 & 0 \\ 2 & 0 \end{bmatrix}$$

13.13.5 The rectangular coordinates of a point: `rectangular_coordinates`

The `rectangular_coordinates` command finds the rectangular coordinates of a point given its polar coordinates.

- `rectangular_coordinates` takes two arguments:
 r , θ , two real numbers.
- `rectangular_coordinates(r, θ)` returns a list of the rectangular coordinates of the point with polar coordinates r and θ .

Example.

Input:

```
rectangular_coordinates(2, pi/4)
```

or:

```
rectangular_coordinates(polar_point(2, pi/4))
```

Output:

$$\left[\frac{2}{2}\sqrt{2}, \frac{2}{2}\sqrt{2} \right]$$

13.13.6 The polar coordinates of a point: `polar_coordinates`

The `polar_coordinates` command finds the polar coordinates of a point.

- `polar_coordinates` takes one argument:
 P , a point.
- `polar_coordinates(P)` returns a list of the polar coordinates of P .

Example.

Input:

```
polar_coordinates(1 + i)
```

or:

```
polar_coordinates(point(1 + i))
```

or:

```
polar_coordinates([1,1])
```

Output:

$$\left[\sqrt{2}, \frac{\pi}{4} \right]$$

13.13.7 The Cartesian equation of a geometric object in the plane: `equation`

See Section 14.12.5 p.1029 for Cartesian equations of three-dimensional objects.

The `equation` command finds the Cartesian equation for a geometric object.

- `equation` takes one argument:
 G , a geometric object.
- `equation(G)` returns the Cartesian equation in the variables x and y for G .

Note that x and y must be formal variables, you might need to purge them with `purge(x)` and `purge(y)`, see Section 5.4.8 p.104.

Example.

Input:

```
equation(line(-1,i))
```

Output:

$$y = x + 1$$

13.13.8 The parametric equation of a geometric object in the plane: `parameq`

See Section 14.12.6 p.1030 for parametric equations in three-dimensional geometry.

The `parameq` command finds a parametric equation for a curve.

- `parameq` takes one argument:
 C , a curve.
- `parameq(G)` returns a parametric equation for G , in the form $x(t) + iy(t)$.

Note that t must be a formal variable, it may be necessary to purge it with `purge(t)`.

Examples.

- *Input:*

```
parameq(line(-1,i))
```

Output:

$$(1 + i)t - 1$$

- *Input:*

```
parameq(circle(-1,i))
```

Output:

$$-1 + ie^{it}$$

• *Input:*

```
normal(parameq(ellipse(-1,1,i)))
```

Output:

$$\frac{-2it^2 - 4t + i}{2t^2 + 1}$$

13.14 Measurements

13.14.1 Measurement and display: `distanceat distanceatraw`
`angleat angleatraw areaat areaatraw perimeterat perimeteratraw`
`slopeat slopeatraw extract_measure`

Many commands to find measures have a version ending in **at** (or **atraw**) which are used to interactively find and display the appropriate measure in a two-dimensional geometry screen. To use them, open a geometry screen with **Alt-G** and then select the appropriate measure from the **Mode ► Measure** menu. Once the mode is selected, then clicking on the names of the appropriate objects (or, if a point is being selected, a name will be automatically generated if clicking on an open point) with the mouse, and then clicking on another point will put the measurement at the point; if the mode is the version ending in **at**, then the measurement will have a label, if the mode is the version ending in **atraw**, then the measurement will appear without a label.

The commands with **at** and **atraw** versions are:

distance, distanceat, distanceatraw This finds the distance between two points or other geometric objects (see Section 13.14.2 p.942).

angle, angleat, angleatraw This finds the measure of an angle BAC given points A , B and C (see Section 13.14.4 p.943).

area, areaat, areaatraw This finds the area of a circle or a polygon which is star-shaped with respect to its first vertex (see Section 13.14.6 p.946).

perimeter, perimeterat, perimeteratraw This finds the perimeter of a circle, circular arc or a polygon (see Section 13.14.7 p.947).

slope, slopeat, slopeatraw This finds the slope of a line, segment, or two points which determine a line (see Section 13.14.8 p.948).

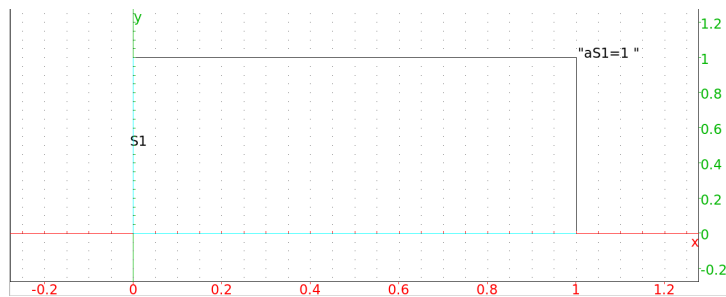
These commands can also be used from the command line. They are like the measurement command but take an extra argument, the point to display the measurement. When using the version ending in **at**, use names for the objects rather create the objects within the measurement command.

Examples.

- *Input:*

```
S1:= square(0,1); areaat(S1,1+i)
```

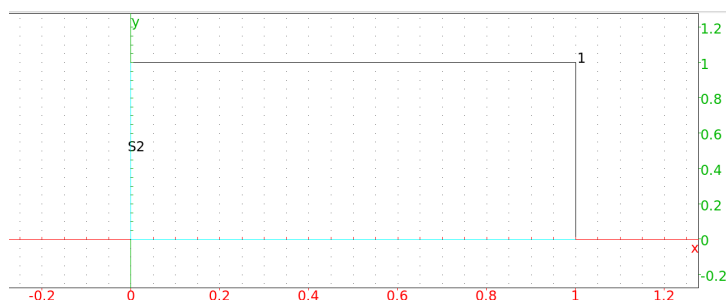
Output:



- *Input:*

```
S2:= square(0,1); areaatraw(S2,1+i)
```

Output:

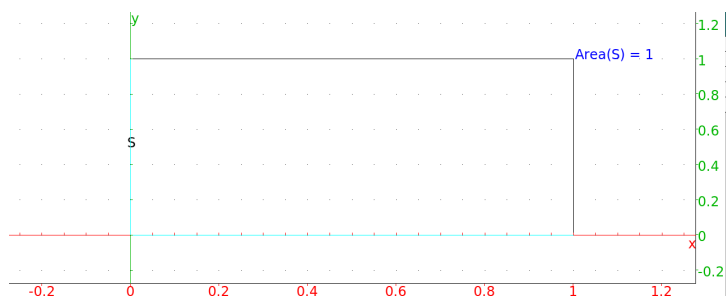


- More sophisticated legends are created with the `legend` command (see Section 13.3.3 p.881).

Input:

```
S:= square(0,1); a:= area(S); legend(1+i,"Area(S) = " +  
string(a),blue)
```

Output:



The `extract_measure` command displays a measurement.

- `extract_measure` takes one argument:
`atcommand`, one of the `at` or `atraw` commands (which displays a measurement).
- `extract_measure(atcommand)` returns the measurement.

Example.

Input:

```
A:= point(-1); B:= point(1+i); C:= point(i);
extract_measure(angleat(A,B,C,0.2i))
```

Output:

$$\arctan\left(\frac{1}{3}\right)$$

13.14.2 The distance between objects in the plane: distance

See Section 14.12.7 p.1030 for distances in three-dimensional geometry.

The `distance` command finds the distance between two geometric objects (a point is considered a geometric object).

- `distance` two arguments:
 G_1, G_2 , two geometric objects.
- `distance(G_1, G_2)` returns the distance between G_1 and G_2 .

Examples.

- *Input:*

```
distance(-1, 1+i)
```

Output:

$$\sqrt{5}$$

- *Input:*

```
distance(0, line(-1,1+i))
```

Output:

$$\frac{\sqrt{5}}{5}$$

- *Input:*

```
distance(circle(0,1),line(-2,1+3*i))
```

Output:

$$\sqrt{2} - 1$$

Note that when the distance calculation uses parameters, `Xcas` must be in real mode.

Example.

In real mode:

Input:

```
assumes(a=[4,0,5,0.1]); A:= point(0); B:= point(a);
simplify(distance(A,B)); simplify(distance(B,A))
```

Output:

$$|a|, |a|$$

In complex mode:

Input:

```
assumes(a=[4,0,5,0.1]); A:= point(0); B:= point(a);
simplify(distance(A,B)); simplify(distance(B,A))
```

Output:

$$-a, a$$

The `distance` command has `distanceat` and `distanceatraw` versions (see Section 13.14.1 p.940).

13.14.3 The length squared of a segment in the plane: `distance2`

See Section 14.12.8 p.1031 for squares of lengths in three-dimensional geometry.

The `distance2` command finds the square of the distance between two points.

- `distance2` takes two arguments:
 P, Q , two points.
- `distance2(P, Q)` returns the square of the distance between P and Q .

Example.

Input:

```
distance2(-1, 1+i)
```

Output:

$$5$$

13.14.4 The measure of an angle in the plane: `angle`

See Section 14.12.9 p.1031 for angle measures in three-dimensional geometry.

The `angle` command finds the measure of an angle.

- `angle` takes three mandatory arguments and one optional argument:
 – A, B, C , three points.

– *str*, a string.

- `angle(A,B,C⟨str⟩)` returns the measure of angle ABC (in the units that Xcas is configured for). With the argument *str*, the angle will be drawn indicated by a small arc and labeled with the string. If the angle is a right angle, the indicator will be a corner rather than an arc.

Examples.

- *Input:*

`angle(0,1,1+i)`

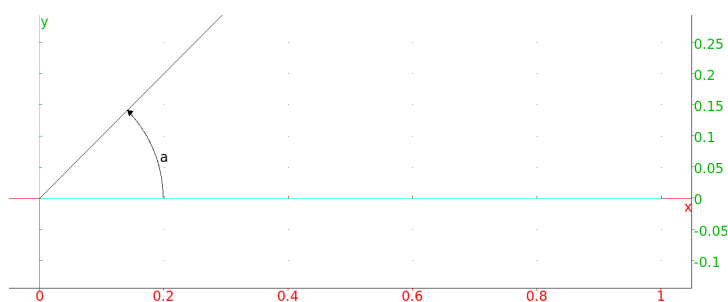
Output:

$$\frac{1}{4}\pi$$

- *Input:*

`angle(0,1,1+i,"a")`

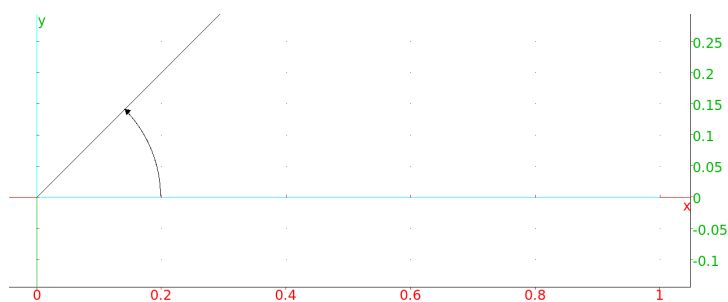
Output:



- *Input:*

`angle(0,1,1+i,"")`

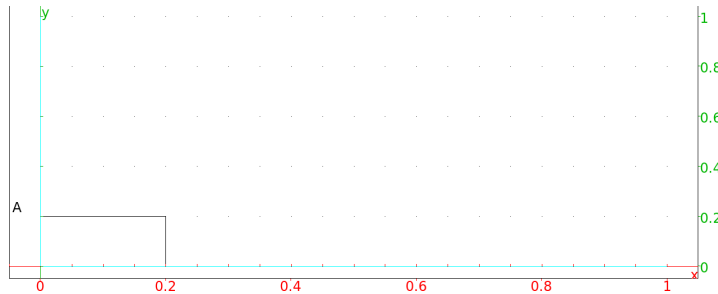
Output:



- *Input:*

`angle(0,1,i,"A")`

Output:



- *Input:*

```
angle(0,1,i,"A")[0]
```

Output:

$$\frac{1}{2}\pi$$

The `angle` command has `angleat` and `angleatraw` versions (see Section 13.14.1 p.940). For the command line versions of these commands, the optional fourth argument for `angle` is replaced by a mandatory fourth argument for the point to put the measurement.

13.14.5 The graphical representation of the area of a polygon: `plotarea` `areaplot`

The `plotarea` finds the (signed) area of a polygon.

`areaplot` is a synonym for `plotarea`.

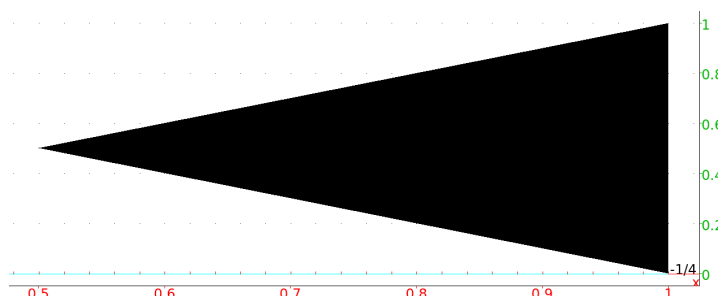
- `plotarea` takes one argument:
 P , a polygon.
- `plotarea(P)` draws the filled polygon, with the signed area. (The area is positive if the polygon is counterclockwise, negative if it is clockwise.)

Examples.

- *Input:*

```
plotarea(polygon(1,(1+i)/2,1+i))
```

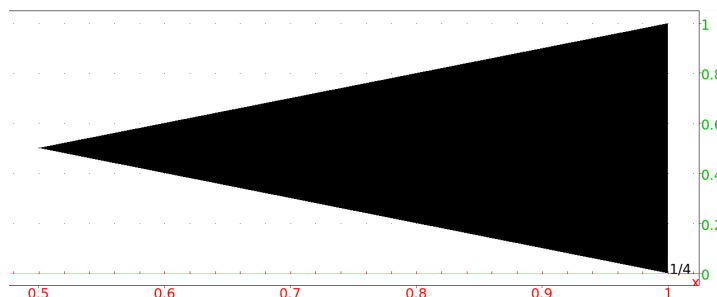
Output:



- *Input:*

```
plotarea(polygon(1,1+i,(1+i)/2))
```

Output:

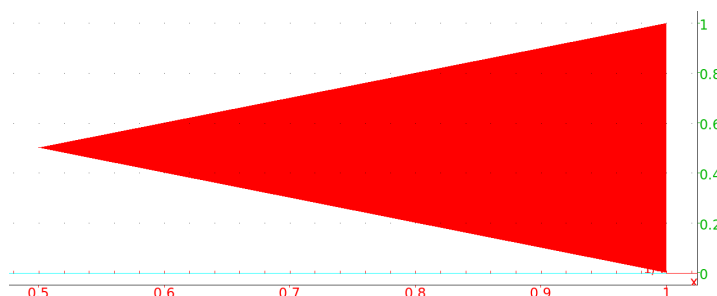


- The fill color can be changed as a local feature (see 13.3.2) and the position of the legend can be changed (see 13.3.3).

Input:

```
plotarea(polygon(1,1+i,(1+i)/2),display=red+quadrant2)
```

Output:



13.14.6 The area of a polygon: area

The **area** command finds the area of a circle or star-shaped polygon.

- **area** takes one argument:
 P , a circle or polygon which is star-shaped with respect to its first vertex (i.e., the line segment from the first vertex to any point in the polygon lies within the polygon).
- **area**(P) returns the area of P .

Examples.

- *Input:*

```
area(triangle(0,1,i))
```

Output:

$$\frac{1}{2}$$

- *Input:*

```
area(square(0,2))
```

Output:

$$4$$

The `area` command has `areaat` and `areaatraw` versions (see Section [13.14.1](#) p.940).

13.14.7 The perimeter of a polygon: `perimeter`

See also `arcLen`, Section [6.19.2](#) p.268.

The `perimeter` command finds the length of a circle, circular arc or polygon.

- `perimeter` takes one argument:
 C , a circle, circular arc or a polygon.
- `perimeter(C)` returns the perimeter of the object.

Examples.

- *Input:*

```
perimeter(circle(0,1))
```

Output:

$$2\pi$$

- *Input:*

```
perimeter(circle(0,1,pi/4,pi))
```

Output:

$$\frac{3}{4}\pi$$

- *Input:*

```
perimeter(arc(0,pi/4,pi))
```

Output:

$$\frac{1}{8}\pi^2$$

- *Input:*

```
perimeter(triangle(0,1,i))
```

Output:

$$\sqrt{2} + 2$$

- *Input:*

```
perimeter(square(0,2))
```

Output:

8

The `perimeter` command has `perimeterat` and `perimeteratraw` versions (see Section [13.14.1](#) p.940).

13.14.8 The slope of a line: `slope`

The `slope` command finds the slope of a line.

- `slope` takes one or two arguments:
 L , a line, a line segment, or two points determining a line.
- `slope(L)` returns the slope of the line.

Examples.

- *Input:*

```
slope(line(1,2i))
```

or:

```
slope(segment(1,2i))
```

or:

```
slope(1,2i)
```

Output:

-2

- *Input:*

```
slope(line(x - 2y = 3))
```

Output:

$$\frac{1}{2}$$

- *Input:*

```
slope(tangent(plotfunc(sin(x)),pi/4))
```

or:

```
slope(LineTan(sin(x),pi/4))
```

Output:

$$\frac{\sqrt{2}}{2}$$

The `slope` command has `slopeat` and `slopeatraw` versions (see Section 13.14.1 p.940).

13.14.9 The radius of a circle: `radius`

The `radius` command finds the radius of a circle.

- `radius` takes one argument:
 C , a circle.
- `radius(C)` returns the radius of C .

Example.

Input:

```
radius(circle(-1,point(i)))
```

Output:

$$\frac{1}{\sqrt{2}}$$

13.14.10 The length of a vector: `abs`

The `abs` command finds the absolute value of a number or the length of a vector (see also Section 6.16.2 p.246 and Section 6.10.4 p.199).

- `abs` takes one argument:
 v , a number or a vector defined by two points.
- `abs(v)` returns the absolute value of v if v is a complex number or the length of v if v is a vector.

Example.

Input:

```
abs(1+i)
```

or:

```
abs(point(1+2*i) - point(i))
```

Output:

$$\sqrt{2}$$

13.14.11 The angle of a vector: `arg`

The `arg` command finds the angle of a complex number (the argument) or the angle of a vector defined by two points.

- `arg` takes one argument:
 v , a number or a vector defined by two points.
- `arg(v)` returns the argument of v if v is a complex number or the angle between the positive x direction and v if v is a vector.

Example.

Input:

```
arg(1+i)
```

Output:

$$\frac{\pi}{4}$$

13.14.12 Normalize a complex number: `normalize`

The `normalize` command normalizes a non-zero complex number; i.e., it finds a complex number with the same argument and absolute value 1.

- `normalize` takes one argument:
 c , a non-zero complex number.
- `normalize(c)` returns the normalized version of c ; namely, $c/|c|$.

Example.

Input:

```
normalize(3+4*i)
```

Output:

$$\frac{3 + 4i}{5}$$

13.15 Transformations**13.15.1 General remarks**

The transformations in this section operate on any geometric object. As arguments, they can take the parameters which specify the transformation. With those arguments, they will return a new command which performs the transformation. If they are given a geometric object as the final argument, they will return the transformed object.

13.15.2 Translations in the plane: translation

See Section 14.14.2 p.1043 for translations in space.

The `translation` command creates a translation.

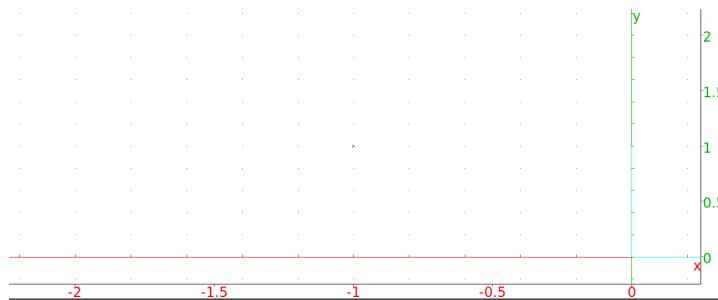
- `translation` takes one mandatory argument and one optional argument:
 - v , the translation vector, which can be given as a vector, a list of coordinates, a difference of points or a complex number.
 - Optionally, G , a geometric object.
- `translation(v)` returns a new command which translates by v .
- `translation(v, G)` returns and draws the translation G by the vector v .

Examples.

- *Input:*

```
t:= translation(1+i)
t(-2)
```

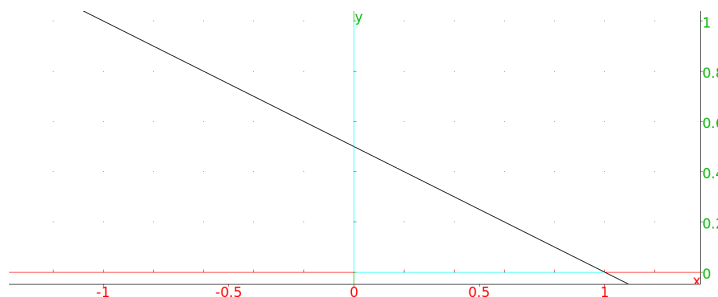
Output:



- *Input:*

```
translation([1,1],line(-2,-i))
```

Output:



13.15.3 Reflections in the plane: reflection

See Section 14.14.3 p.1045 for reflections in space.

The `reflection` command creates a reflection.

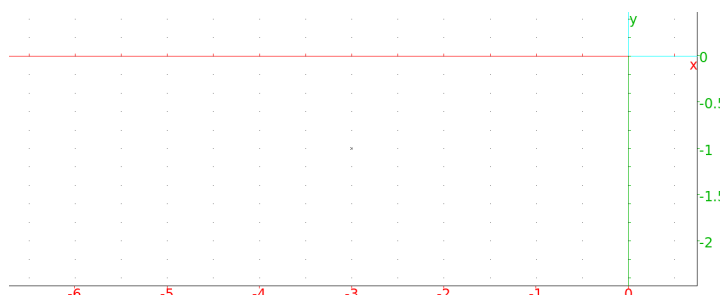
- `reflection` takes one mandatory argument and one optional argument:
 - P , a point or a line.
 - Optionally, G , a geometric object.
- `reflection(P)` returns a new command which reflects about P .
- `reflection(P, G)` returns and draws the reflection of G about P .

Examples.

- *Input:*

```
rf:= reflection(-1)
rf(1+i)
```

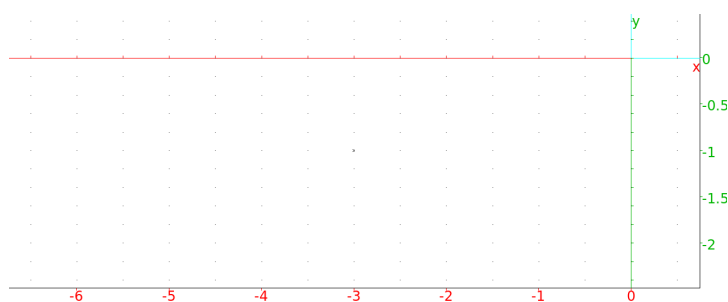
Output:



- *Input:*

```
reflection(-1, 1+i)
```

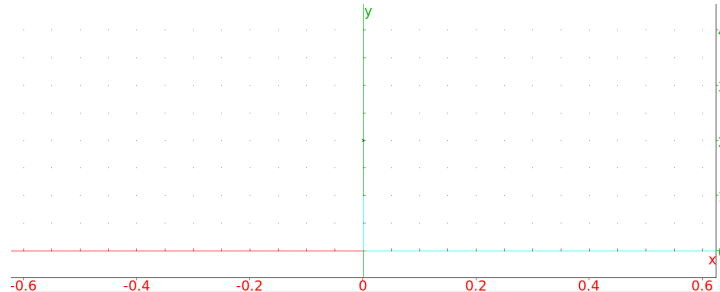
Output:



Input:

```
reflection(line(-1,i),1+i)
```


Output:



13.15.4 Rotation in the plane: rotation

See Section 14.14.4 p.1046 for rotations in space.

The `rotation` command creates a rotation.

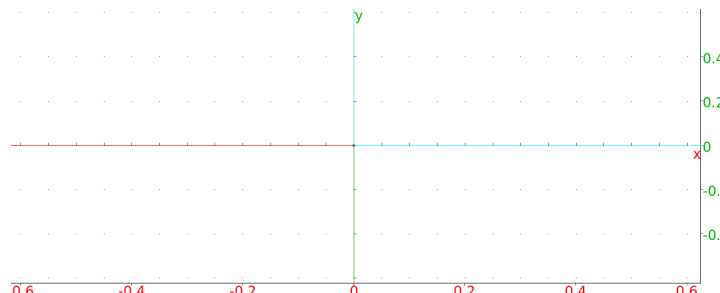
- `rotation` takes two mandatory arguments and one optional argument:
 - P , a point (the center of rotation).
 - θ , the angle of rotation.
 - Optionally, G , a geometric object.
- `rotation(P, θ)` returns a new command which rotates about P through an angle of θ .
- `reflection(P, θ, G)` returns and draws the rotation of G about P through an angle of θ .

Examples.

- *Input:*

```
r:= rotation(i, -pi/2)
r(1+i)
```

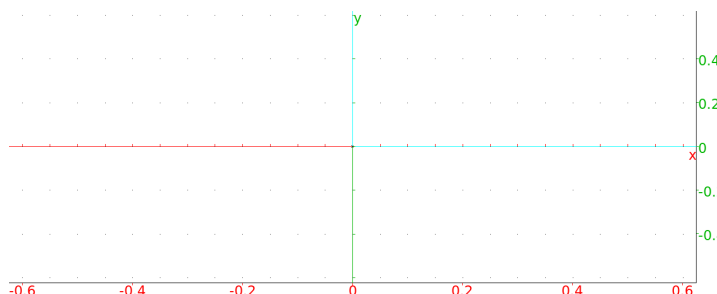
Output:



- *Input:*

```
rotation(i, -pi/2, 1+i)
```

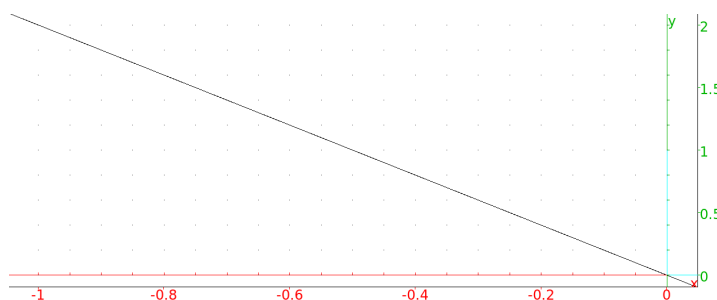
Output:



Input:

```
rotation(i, -pi/2, line(1+i, -1))
```

Output:



13.15.5 Homothety in the plane: homothety

See Section 14.14.5 p.1047 for homotheties in space.

A homothety is a dilation about a given point. The `homothety` command creates a homothety.

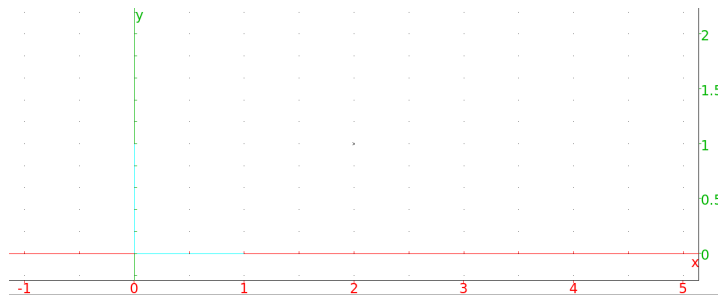
- `homothety` takes two mandatory arguments and one optional argument:
 - P , a point (the center of the homothety).
 - r , a number (the scaling ratio).
 - Optionally, G , a geometric object.
- `homothety(P, r)` returns a new command which dilates about P by a factor of r . If r is complex, this will rotate as well as scale.
- `homothety(P, r, G)` returns and draws the dilation of G about P by a factor or r .

Examples.

- *Input:*

```
h:= homothety(i, 2)
h(1+i)
```

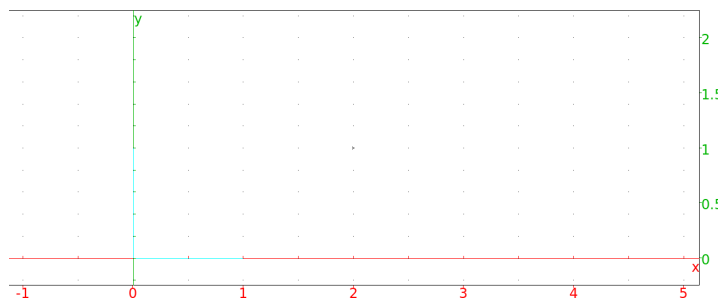
Output:



• *Input:*

```
homothety(i, 2, 1+i)
```

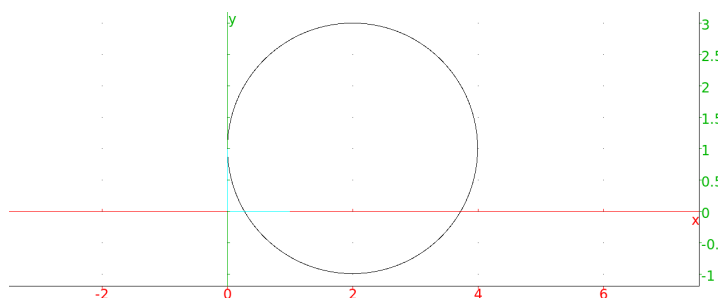
Output:



• *Input:*

```
homothety(i, 2, circle(1+i,1))
```

Output:



13.15.6 Similarity in the plane: similarity

See Section 14.14.6 p.1048 for similarities in space.

The **similarity** command creates a command to rotate and scale about a given point.

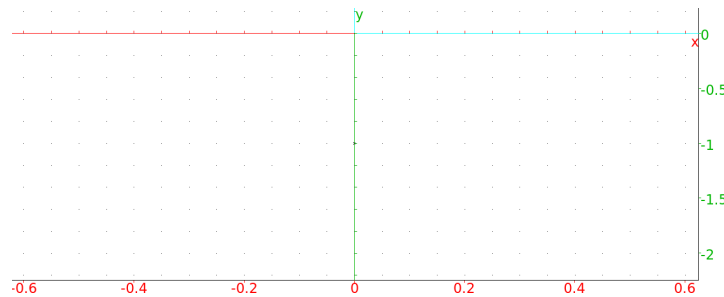
- **similarity** takes three mandatory arguments and one optional argument:

- P , a point (the center of the rotation).
 - r , a real number (the scaling ratio).
 - θ , a real number (the angle of rotation).
 - Optionally, G , a geometric object.
- `similarity(P, r, θ)` returns a new command which rotates about P through an angle of θ and scales about P by a factor of r .
 - `similarity(P, r, θ, G)` returns and draws the transformation of G .

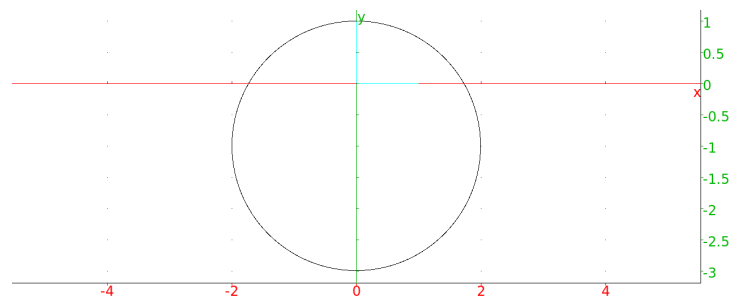
Examples.

- *Input:*

```
s:= similarity(i, 2, -pi/2)
s(1+i)
```

Output:*then:*

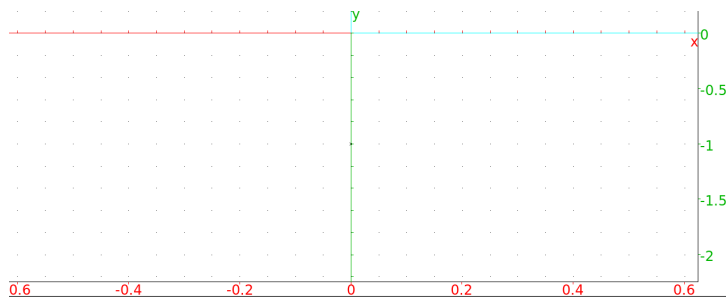
```
s(circle(1+i,1))
```

Output:

- *Input:*

```
similarity(i, 2, -pi/2, 1 + i)
```

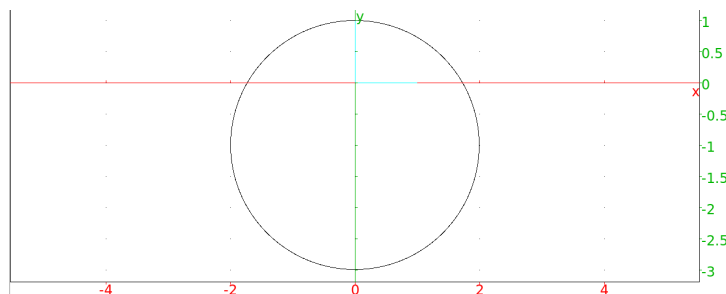
Output:



- *Input:*

```
similarity(i, 2, -pi/2, circle(1+i,1))
```

Output:



Note that for a point P and real numbers r and θ , the command `similarity(P, r, θ)` is the same as `homothety($P, k \cdot \exp(i \cdot a)$)`.

13.15.7 Inversion in the plane: inversion

See Section 14.14.7 p.1049 for inversions in space.

Given a circle C with center P and radius r , the *inversion* of a point A with respect to C is the point A' on the ray \overrightarrow{PA} satisfying $\overline{PA} \cdot \overline{PA'} = r^2$.

The `inversion` command creates an inversion.

- `inversion` takes two mandatory arguments and one optional argument:
 - P , a point (the center of the inversion).
 - r , a real number (the radius).
 - Optionally, G , a geometric object.
- `inversion(P, r)` returns a new command which performs the inversion.
- `inversion(P, r, G)` returns and draws the inversion of G .

Examples.

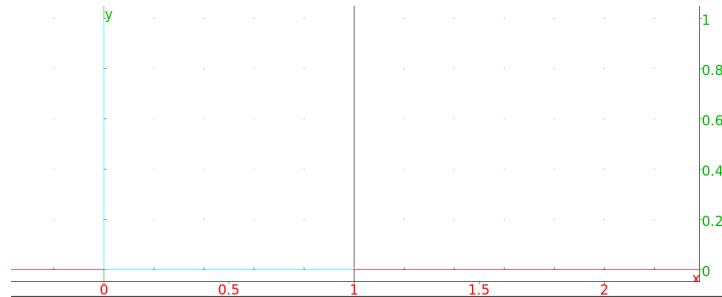
- *Input:*

```

inver:= inversion(i, 2)
inver(circle(1+i,1))

```

Output:



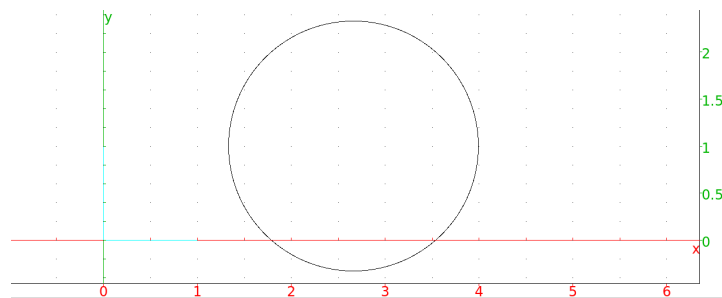
then:

```

inver(circle(1+i,1/2))

```

Output:



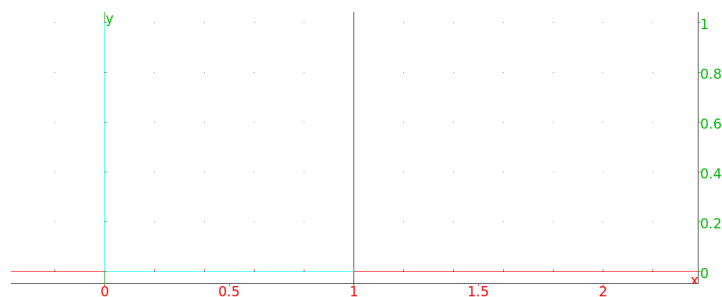
• *Input:*

```

inversion(i, 2, circle(1+i,1))

```

Output:



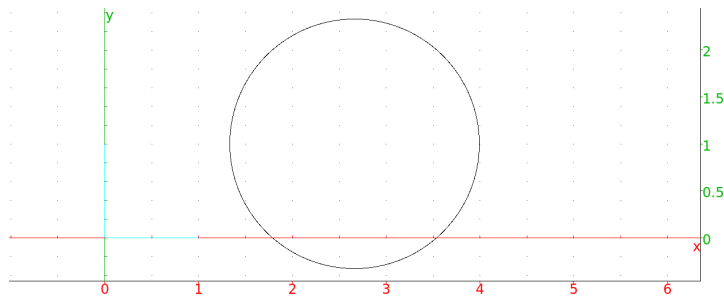
Input:

```

inversion(i, 2, circle(1+i,1/2))

```

Output:



13.15.8 Orthogonal projection in the plane: projection

See Section 14.14.8 p.1051 for projections in space.

The `projection` command creates a projection.

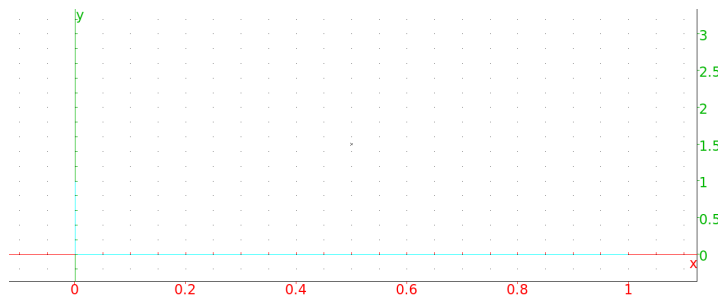
- `projection` takes one mandatory argument and one optional argument:
 - O , a geometric object.
 - Optionally, G , a geometric object.
- `projection(O)` returns a new command which projects points onto O .
- `projection(O, G)` returns and draws the projection of G onto O .

Examples.

- *Input:*

```
p1:= projection(line(-1,i))
p1(i+1)
```

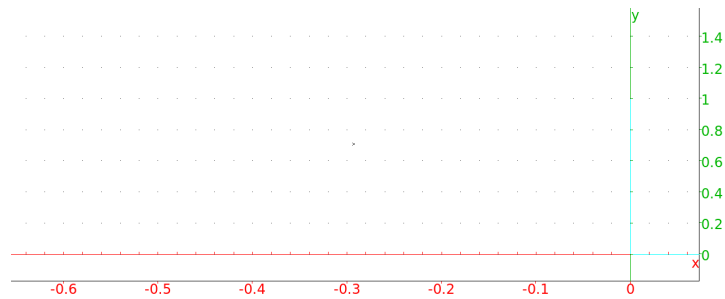
Output:



- *Input:*

```
p2:= projection(circle(-1,1))
p2(i)
```

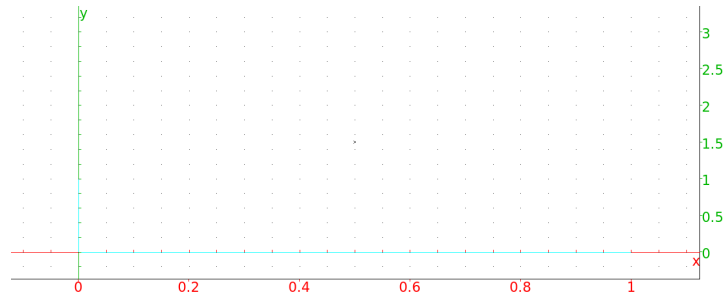
Output:



- *Input:*

```
projection(line(-1, i), 1+i)
```

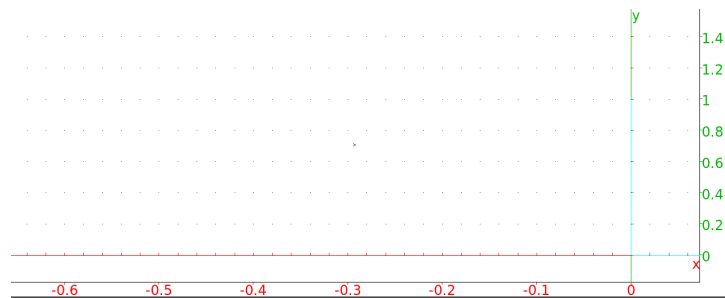
Output:



- *Input:*

```
projection(circle(-1,1), i)
```

Output:



13.16 Properties

13.16.1 Checking if a point is on an object in the plane: `is_element`

See Section [14.13.1](#) p.1032 for checking elements in three-dimensional geometry.

The `is_element` command determines whether or not a point is on a geometric object.

- `is_element` takes two arguments:
 - P , a point.
 - G , a geometric object.
- `is_element(P, G)` returns 1 if P is an element of G and returns 0 otherwise.

Examples.

- *Input:*

```
is_element(-1-i, line(0,1+i))
```

Output:

1

- *Input:*

```
is_element(i, line(0,1+i))
```

Output:

0

13.16.2 Checking if three points are collinear in the plane:

`is_collinear`

See Section 14.13.6 p.1036 for checking for collinearity in three-dimensional geometry.

The `is_collinear` command determines whether or not three points are collinear.

- `is_collinear` takes three arguments:
 - A, B, C , three points.
- `is_collinear(A, B, C)` returns 1 if A, B and C are collinear and returns 0 otherwise.

Examples.

- *Input:*

```
is_collinear(0,1+i,-1-i)
```

Output:

1

- *Input:*

```
is_collinear(i/100, 1+i, -1-i)
```

Output:

0

13.16.3 Checking if four points are concyclic in the plane: `is_concyclic`

See Section 14.13.7 p.1037 for checking for concyclicity in three-dimensional geometry.

The `is_concyclic` command determines whether or not points are cyclic.

- `is_concyclic` takes one argument:
 L , a list or sequence of points.
- `is_concyclic(L)` returns 1 if the points in L all lie on the same circle, and returns 0 otherwise.

Examples.

- *Input:*

```
is_concyclic(1+i, -1+i, -1-i, 1-i)
```

Output:

1

- *Input:*

```
is_concyclic(i, -1+i, -1-i, 1-i)
```

Output:

0

13.16.4 Checking if a point is in a polygon or circle: `is_inside`

The `is_inside` command determines whether or not a point is in a polygon or a circle.

- `is_inside` takes two arguments:
 - P , a point.
 - C , a polygon or a circle.
- `is_inside(P, C)` returns 1 if P is inside C (including the boundary) and returns 0 otherwise.

Examples.

- *Input:*

```
is_inside(0, circle(-1,1))
```

Output:

1

- *Input:*

```
is_inside(2,polygon([1,2-i,3+i]))
```

Output:

1

- *Input:*

```
is_inside(1-i, triangle([1,2-i,3+i]))
```

Output:

0

13.16.5 Checking if an object is an equilateral triangle in the plane: `is_equilateral`

See Section [14.13.9](#) p.1038 for checking for equilateral triangles in three-dimensional geometry.

The `is_equilateral` command determines whether or not a geometric object is an equilateral triangle.

- `is_equilateral` takes one argument:
 G , a geometric object or a sequence of three points assumed to be the vertices of a triangle.
- `is_equilateral(G)` returns 1 if the object is an equilateral triangle and returns 0 otherwise.

Examples.

- *Input:*

```
is_equilateral(0,2,1+i*sqrt(3))
```

Output:

1

- *Input:*

```
T:= equilateral_triangle(0,2,C)
is_equilateral(T[0])
```

Output:

1

Note that `T[0]` is a triangle since `T` is a list made of a triangle and the vertex `C`.

Input:

```
affix(C)
```

Output:

$$\sqrt{3}i + 1$$

• *Input:*

```
is_equilateral(1+i, -1+i, -1-i)
```

Output:

0

13.16.6 Checking if an object in the plane is an isosceles triangle: `is_isosceles`

See Section 14.13.10 p.1039 for checking for isosceles triangles in three-dimensional geometry.

The `is_isosceles` command determines whether or not a geometric object is an isosceles triangle.

- `is_isosceles` takes one argument:
 G , a geometric object or a sequence of three points assumed to be the vertices of a triangle.
- `is_isosceles(G)` returns 1, 2 or 3 if the object is an isosceles triangle (the number indicates which vertex is on two equal sides), returns 4 if the object is an equilateral triangle, and returns 0 otherwise.

Examples.

• *Input:*

```
is_isosceles(0, 1+i,i)
```

Output:

2

• *Input:*

```
T:= isosceles_triangle(0,1,pi/4)
is_isosceles(T)
```

Output:

1

• *Input:*

```
T:= isosceles_triangle(0,1,pi/4,C)
is_isosceles(T[0])
```

Output:

1

Note that `T[0]` is a triangle since `T` is a list made of a triangle and the vertex `C`.

Input:

`affix(C)`

Output:

$$\frac{\sqrt{2}}{2} + \frac{\sqrt{2}}{2}i$$

- *Input:*

`is_isosceles(1+i, -1+i, -i)`

Output:

3

- *Input:*

`is_isosceles(0,2,1+i*sqrt(3))`

Output:

4

13.16.7 Checking if an object in the plane is a right triangle or a rectangle: `is_rectangle`

See Section 14.13.11 p.1040 for checking for right triangles and rectangles in three-dimensional geometry.

The `is_rectangle` command determines whether or not a geometric object is an rectangle or a right triangle.

- `is_rectangle` takes one argument:
 G , a geometric object or a sequence of three or four points assumed to be the vertices of a triangle or a quadrilateral.
- `is_rectangle(G)` returns:
 - (for triangle G) 1, 2 or 3 if G is a right triangle (the number indicates which vertex is has the right angle).
 - (for quadrilaterals G) 1 if G is a rectangle but not a square.
 - (for quadrilaterals G) 2 if G is square.
 - 0 otherwise.

Example.

Input:

`is_rectangle(1,1+i,i)`

Output:

2

- *Input:*

```
is_rectangle(1+i, -2+i, -2-i, 1-i)
```

Output:

1

- *Input:*

```
R:= rectangle(-2-i,1-i, 3, C, D)
is_rectangle(R[0])
```

Output:

1

Note that $R[0]$ is a rectangle since R is a list made of a rectangle and vertices C and D .

Input:

```
affix(C,D)
```

Output:

$-2 + 8i, 1 + 8i$

13.16.8 Checking if an object in the plane is a square: `is_square`

See Section 14.13.12 p.1040 for checking for squares in three-dimensional geometry.

The `is_square` command determines whether or not a geometric object is a square.

- `is_square` takes one argument:
 G , a geometric object or a sequence of four points assumed to be the vertices of a quadrilateral.
- `is_square(G)` returns 1 if the object is a square and returns 0 otherwise.

Examples.

- *Input:*

```
is_square(1+i, -1+i, -1-i, 1-i)
```

Output:

1

- *Input:*

```
K:= square(1+i, -1+i)
is_square(K)
```

Output:

1

- *Input:*

```
K:= square(1+i, -1+i, C, D)
is_square(K[0])
```

Output:

1

Note that $K[0]$ is a square since K is a list made of a square and vertices C and D .

Input:

```
affix(C,D)
```

Output:

$-1 - i, 1 - i$

- *Input:*

```
is_square(i, -1+i, -1-i, 1-i)
```

Output:

0

13.16.9 Checking if an object in the plane is a rhombus: `is_rhombus`

See Section 14.13.13 p.1041 for checking for rhombuses in three-dimensional geometry.

The `is_rhombus` command determines whether or not a geometric object is a rhombus.

- `is_rhombus` takes one argument:
 G , a geometric object or a sequence of four points assumed to be the vertices of a quadrilateral.
- `is_square(G)` returns 1 if G is a rhombus but not a square, returns 2 if G is a square and returns 0 otherwise.

Examples.

- *Input:*

```
is_rhombus(1+i, -1+i, -1-i, 1-i)
```

Output:

1

- *Input:*

```
K:= rhombus(1+i, -1+i, pi/4)
is_rhombus(K)
```

Output:

1

- *Input:*

```
K:= rhombus(1+i, -1+i, pi/4, C, D)
is_rhombus(K[0])
```

Output:

1

Note that $K[0]$ is a rhombus since K is a list made of a rhombus and vertices C and D .

Input:

```
affix(C,D)
```

Output:

$-\sqrt{2} - i, -\sqrt{2} + i$

- *Input:*

```
is_rhombus(i, -1+i, -1-i, 1-i)
```

Output:

0

13.16.10 Checking if an object in the plane is a parallelogram: `is_parallelogram`

See Section 14.13.14 p.1042 for checking for parallelograms in three-dimensional geometry.

The `is_parallelogram` command determines whether or not an object is a parallelogram.

- `is_parallelogram` takes one argument:
 G , a geometric object or a sequence of four points assumed to be the vertices of a quadrilateral.
- `is_parallelogram(G)` returns 1 if G is a parallelogram, but not a rhombus or a rectangle, returns 2 if G is a rhombus but not a rectangle, returns 3 if G is a rectangle but not a square, returns 4 if G is a square, and returns 0 otherwise.

Examples.

- *Input:*

```
is_parallelogram(i, -1+i, -1-i, 1-i)
```


Output:

0

- *Input:*

```
is_parallelogram(1+i, -1+i, -1-i, 1-i)
```

Output:

1

- *Input:*

```
Q:= quadrilateral(1+i, -1+i, -1-i, 1-i)
is_parallelogram(Q)
```

Output:

4

- *Input:*

```
P:= parallelogram(-1-i, 1-i, i, D)
is_parallelogram(P[0])
```

Output:

1

Note that $P[0]$ is a parallelogram since P is a list made of a parallelogram and vertex D .

Input:

```
affix(D)
```

Output:

$-2 + i$

13.16.11 Checking if two lines in the plane are parallel: `is_parallel`

See Section 14.13.3 p.1034 for checking for parallels in three-dimensional geometry.

The `is_parallel` command determines whether or not two lines are parallel.

- `is_parallel` takes two arguments:
 L_1, L_2 , two lines.
- `is_parallel(L_1, L_2)` returns 1 if L_1 and L_2 are parallel and otherwise returns 0.

Examples.

- *Input:*

```
is_parallel(line(0,1+i),line(i,-1))
```

Output:

1

- *Input:*

```
is_parallel(line(0,1+i),line(i,-1-i))
```

Output:

0

13.16.12 Checking if two lines in the plane are perpendicular: `is_perpendicular`

See Section 14.13.4 p.1035 for checking for perpendicularity in three-dimensional geometry.

The `is_perpendicular` command determines whether or not two lines are perpendicular.

- `is_perpendicular` takes two arguments:
 L_1, L_2 , two lines.
- `is_perpendicular(L_1, L_2)` returns 1 if L_1 and L_2 are perpendicular and otherwise returns 0.

Examples.

- *Input:*

```
is_perpendicular(line(0,1+i),line(i,1))
```

Output:

1

- *Input:*

```
is_parallel(line(0,1+i),line(1+i,1))
```

Output:

0

13.16.13 Checking if two circles in the plane are orthogonal: `is_orthogonal`

See Section 14.13.5 p.1035 for checking for orthogonality in three-dimensional geometry.

The `is_orthogonal` command determines whether or not two lines or circles are orthogonal.

- `is_orthogonal` takes two arguments:
 C_1, C_2 , two objects, both lines or both circles.

- `is_orthogonal(C_1, C_2)` returns 1 if C_1 and C_2 are orthogonal and returns 0 otherwise.

Examples.

- *Input:*

```
is_orthogonal(line(0,1+i),line(i,1))
```

Output:

1

- *Input:*

```
is_orthogonal(line(2,i),line(0,1+i))
```

Output:

0

- *Input:*

```
is_orthogonal(circle(0,1+i),circle(2,1+i))
```

Output:

1

- *Input:*

```
is_orthogonal(circle(0,1),circle(2,1))
```

Output:

0

13.16.14 Checking if elements are conjugates: `is_conjugate`

The `is_conjugate` command determines whether or not two objects are conjugates.

To check for conjugates with respect to a circle:

- `is_conjugate` takes three arguments:
 - C , a circle.
 - P, Q , each of which is a point or a line.
- `is_conjugate(C, P, Q)` returns 1 if P and Q are conjugate with respect to C , otherwise it returns 0.

Examples.

- *Input:*

```
is_conjugate(circle(0,1+i),point(1-i), point(3+i))
```

Output:

1

- *Input:*

```
is_conjugate(circle(0,1),point((1+i)/2), line(1+i,2))
```

Output:

1

- *Input:*

```
is_conjugate(circle(0,1), line(1+i,2), line((1+i)/2,0))
```

Output:

1

To check for conjugates with respect to two points or two lines:

- `is_conjugate` takes three arguments:
 - L_1, L_2 , two lines or two points.
 - P, Q , each of which is a point or a line.
- `is_conjugate(L_1, L_2, P, Q)` returns 1 if P and Q are conjugate with respect to L_1 and L_2 , otherwise it returns 0.

Examples.

- *Input:*

```
is_conjugate(point(1+i),point(3+i),point(i),point(3/2+i))
```

Output:

1

- *Input:*

```
is_conjugate(line(0,1+i),line(2,3+i),line(3,4+i),line(3/2,5/2+i))
```

Output:

1

13.16.15 Checking if four points form a harmonic division: `is_harmonic`

The `is_harmonic` command determines whether or not four points form a harmonic division.

- `is_harmonic` takes four arguments:
 P, Q, R, S , four points.
- `is_harmonic(P, Q, R, S)` returns 1 if P, Q, R and S form a harmonic range and returns 0 otherwise.

Examples.

- *Input:*

```
is_harmonic(0,2,3/2,3)
```

Output:

1

- *Input:*

```
is_harmonic(0,1+i,1,i)
```

Output:

0

13.16.16 Checking if lines are in a bundle: `is_harmonic_line_bundle`

The `is_harmonic_line_bundle` command determines how lines are related.

- `is_harmonic_line_bundle` takes one argument:
 L , a list or sequence of lines.
- `is_harmonic_line_bundle(L)` returns:
 - 1 if the lines pass through a common point
 - 2 if the lines are parallel
 - 3 if the lines are the same
 - 0 otherwise

Example.

Input:

```
is_harmonic_line_bundle([line(0,1+i),line(0,2+i),line(0,3+i),line(0,1)])
```

Output:

1

13.16.17 Checking if circles are in a bundle: `is_harmonic_circle_bundle`

The `is_harmonic_circle_bundle` command determines how circles are related.

- `is_harmonic_circle_bundle` takes one argument:
 L , a list or sequence of circles.
- `is_harmonic_circle_bundle(L)` returns:
 - 1 if the circles pass through a common point.
 - 2 if the circles are concentric.
 - 3 if the circles are the same.
 - 0 otherwise.

Example.

Input:

```
is_harmonic_circle_bundle([circle(0,i),circle(4,i),circle(0,1/2)])
```

Output:

1

13.17 Harmonic division**13.17.1 Finding a point dividing a segment in the harmonic ratio k : `division_point`**

The `division_point` command finds a point dividing a segment in a given ratio.

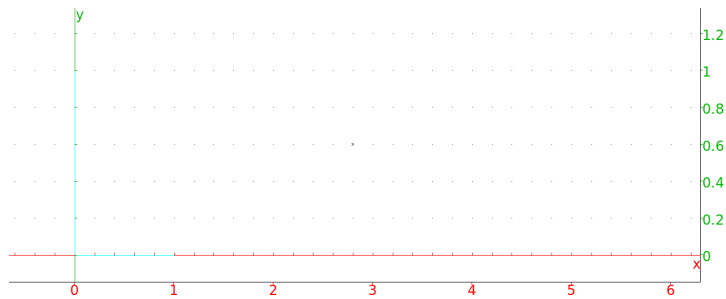
- `division_point` takes three arguments:
 - a, b , two complex numbers or points.
 - k , a complex number.
- `division_point(a, b, k)` returns and draws z where $(z-a)/(z-b) = k$.

Examples.

- *Input:*

```
division_point(i,2+i,3+i)
```

Output:



- *Input:*

```
affix(division_point(i,2+i,3))
```

Output:

$$3 + i$$

13.17.2 The cross ratio of four collinear points: `cross_ratio`

The cross ratio of four numbers a, b, c, d is $[(c - a)/(c - b)]/[(d - a)/(d - b)]$.

The `cross_ratio` command finds the cross ratio.

- `cross_ratio` takes for arguments:
 a, b, c, d , complex numbers.
- `cross_ratio(a, b, c, d)` returns the cross ratio, $[(c - a)/(c - b)]/[(d - a)/(d - b)]$.

Examples.

- *Input:*

```
cross_ratio(0,1,2,3)
```

Output:

$$\frac{4}{3}$$

- *Input:*

```
cross_ratio(i,2+i,3/2 + i, 3+i)
```

Output:

$$-1$$

13.17.3 Harmonic division: `harmonic_division`

Four collinear points A, B, C and D are in harmonic division if $\overline{CA}/\overline{CB} = \overline{DA}/\overline{DB}$. In this case, D is called the harmonic conjugate of A, B and C .

Four concurrent lines or four parallel lines are in harmonic division if the intersection of any fifth line with these four lines consists of four points in harmonic division. The lines are also said to form a harmonic pencil. The fourth line is called the harmonic conjugate of the first three.

The `harmonic_division` command finds harmonic conjugates.

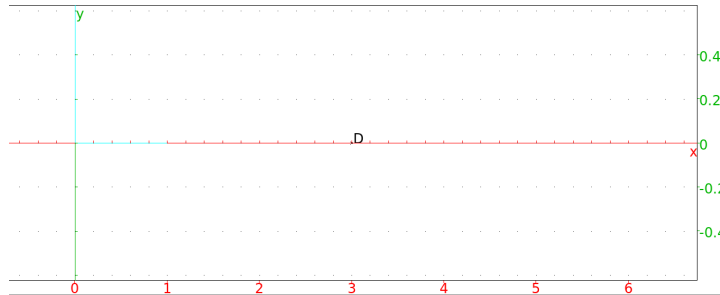
- `harmonic_division` takes four arguments:
 - P_1, P_2, P_3 , three collinear points or three concurrent lines.
 - *var*, a variable name.
- `harmonic_division(P_1, P_2, P_3, var)` returns and draws the three points or lines P_1, P_2 and P_3 with a fourth so the four objects are in harmonic division, and assigns the fourth point or line to *var*.

Examples.

- *Input:*

```
harmonic_division(0,2,3/2,D)
```

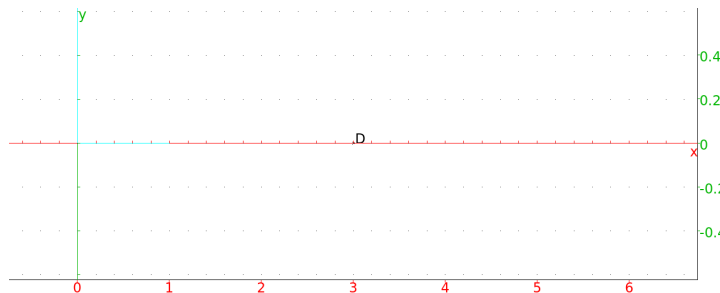
Output:



- *Input:*

```
harmonic_division(point(0),point(2),point(3/2),D)
```

Output:



Input:

```
affix(D)
```

Output:

3

13.17.4 The harmonic conjugate: harmonic_conjugate

The `harmonic_conjugate` command finds harmonic conjugates.

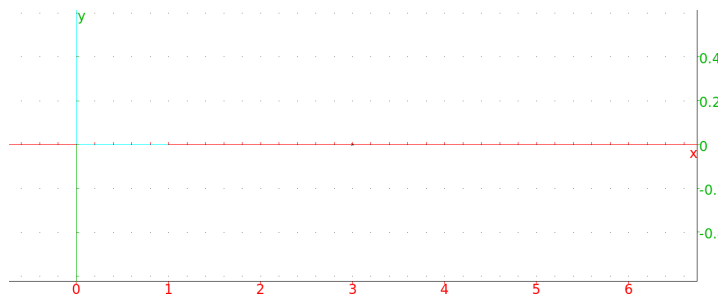
- *harmonic_conjugate* takes three arguments:
- P_1, P_2, P_3 , three collinear points or three concurrent lines, or three parallel lines.
- `harmonic_conjugate(P_1, P_2, P_3)` returns and draws the harmonic conjugate of P_1, P_2 and P_3 .

Examples.

- *Input:*

```
harmonic_conjugate(0,2,3/2)
```

Output:



- *Input:*

```
affix(harmonic_conjugate(0,2,3/2))
```

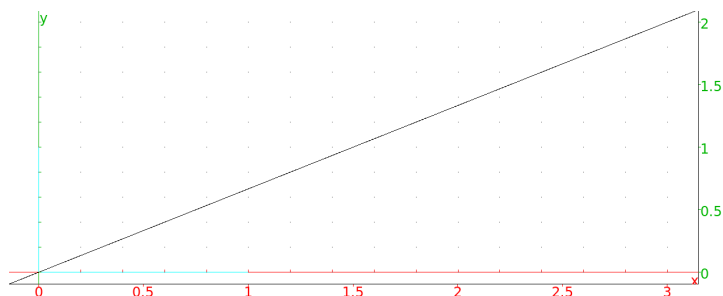
Output:

3

- *Input:*

```
harmonic_conjugate(line(0,1+i),line(0,3+i),line(0,i))
```

Output:



13.17.5 Pole and polar: pole polar

Given a circle centered at O , a point A is a pole and a line L is the corresponding polar if L is the line passing through the inversion of A with respect to the circle (see Section 13.15.7 p.957) passing through the line \overleftrightarrow{OA} .

The `polar` command finds the polar of a point.

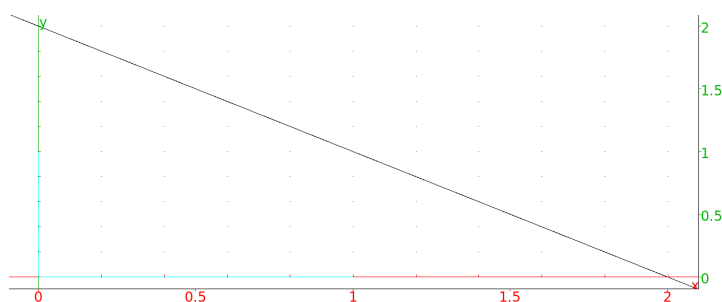
- `polar` takes two arguments:
 - C , a circle.
 - A , a point.
- `polar(C, A)` returns and draws the polar of the point A with respect to C .

Example.

Input:

```
polar(circle(0,1),(i+1)/2)
```

Output:



The `pole` command finds the pole of a line.

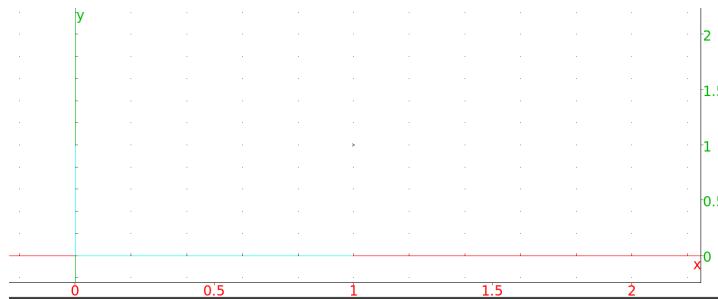
- `pole` takes two arguments:
 - C , a circle.
 - L , a line
- `pole(C, L)` returns and draws the pole of the line L with respect to C .

Examples.

- *Input:*

```
pole(circle(0,1),line(i,1))
```

Output:



- *Input:*

```
affix(pole(circle(0,1),line(i,1)))
```

Output:

$$1 + i$$

13.17.6 The polar reciprocal: reciprocation

The **reciprocation** command finds poles and polars.

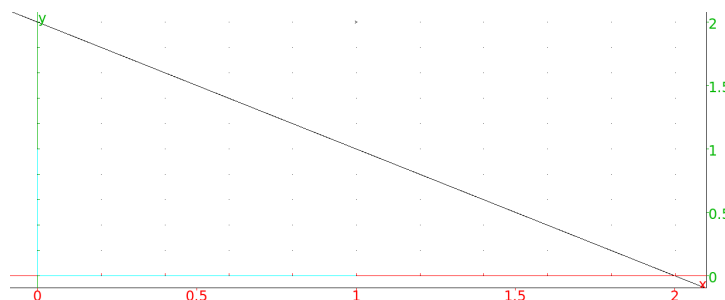
- *reciprocation* takes two arguments:
 - C , a circle.
 - L , a list of points and lines.
- **reciprocation**(C,L) returns the list formed by replacing each point or line in L by its polar or pole with respect to the circle C .

Example.

Input:

```
reciprocation(circle(0,1),[point((1+i)/2),line(1,-1+i)])
```

Output:



13.18 Loci and envelopes

13.18.1 Loci: locus

The `locus` command draws the locus of points determined by geometric objects moving in the plane, where the object depends on a point moving along a curve. It can draw a locus of points which depends on points on a curve, or the envelope of a family of lines depending on points on a curve.

The locus of points depending on points on a curve.

For drawing the locus of points depending on points on a curve:

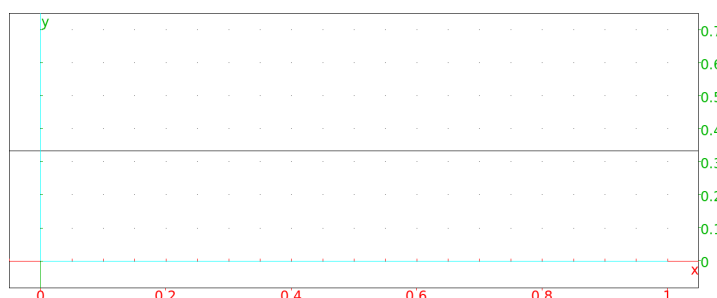
- `locus` takes two mandatory arguments and two optional arguments:
 - var_1 , a variable name which has already been assigned to a point, which itself is a function of var_2 , the second argument.
 - var_2 , a variable name which is assigned to `element(C)` for some curve C (see Section 13.6.15 p.897).
 - Optionally, $t = a..b$, where t is the parameter of the curve C . (You can double check the name of the parameter for a curve C with the command `parameq(C)`.)
 - Optionally, `tstep=s`, to set the step size for the parameter t .
- `locus(var_1, var_2 {tstep=c})` draws the locus of points formed by var_1 , as var_2 traces over the curve C .
 With the optional arguments, C is limited to the part parameterized from a to b , with a step size of c .

Examples.

- *Input:*

```
P:= element(line(i, i+1))
G:= isobarycenter(-1,1,P)
locus(G,P)
```

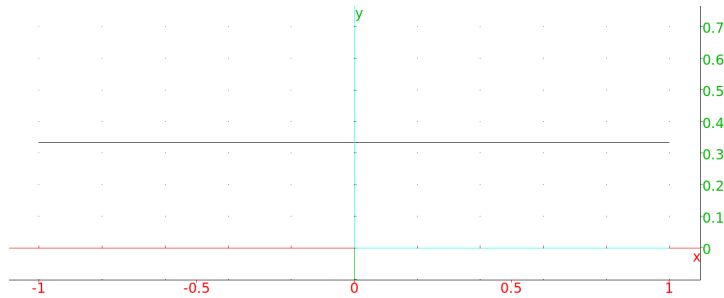
This will draw the set of isobarycenters of the triangles with vertices $-1, 1$ and P , where P ranges over the line through i and $i+1$. *Output:*



- *Input:*

```
locus(G,P,t=-3..3,tstep=0.1)
```

Output:



The envelope of a family of lines which depend on points on a curve.

For drawing the envelope of a family of lines which depend on points on a curve:

- **locus** takes two mandatory arguments and two optional arguments.
 - var_1 , a variable name which has already been assigned to a line, which itself is a function of var_2 , the second argument.
 - var_2 , a variable name which has already been assigned to **element**(C) for some curve C (see Section 13.6.15 p.897).
 - Optionally, $t = a..b$, where t is the parameter of the curve C . (You can double check the name of the parameter for a curve C with the command **parameq**(C).)
 - Optionally, **tstep**= s , to set the step size for the parameter t .
- **locus**(var_1, var_2 **tstep**= c) draws the envelope of lines formed by var_1 , as var_2 traces over the curve C .
 With the optional arguments, C is limited to the part parameterized from a to b , with a step size of c .

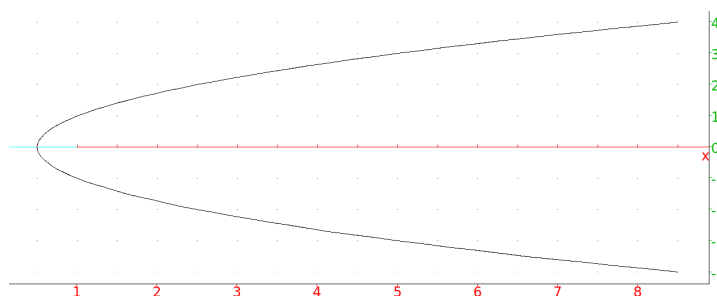
Examples.

- *Input:*

```
F:= point(1)
H:= element(line(x=0))
d:= perpen_bisector(F,H)
locus(d,H)
```

This will draw the envelope of the family of perpendicular bisectors of the segments from the point 1 to the points on the line $x=0$.

Output:



- To draw the envelope of a family of lines which depend on a parameter, such as the lines given by the equations

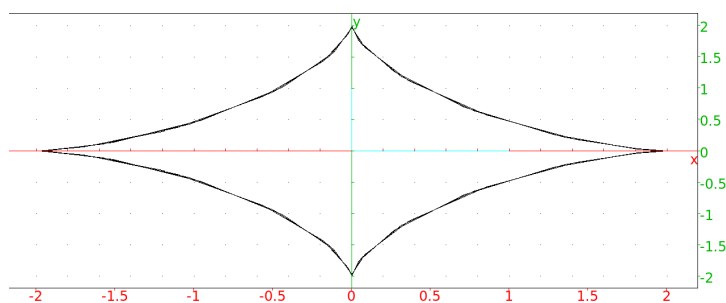
$$y + x \tan(t) - 2 \sin(t) = 0$$

over the parameter t , the parameter can be regarded as the affixes of points on the line $y = 0$.

Input:

```
H:= element(line(y=0))
D:= line(y + x*tan(affix(H)) - 2*sin(affix(H)))
locus(D,H)
```

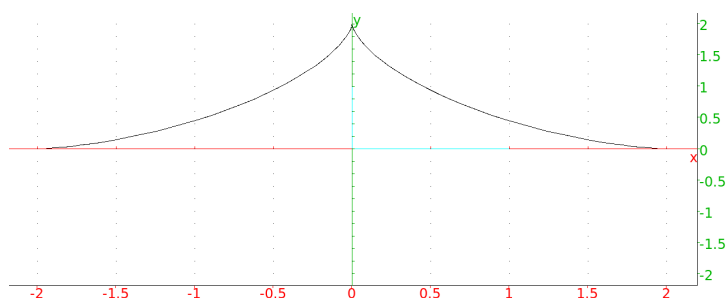
Output:



- *Input:*

```
locus(D,H,t=0..pi)
```

Output:



13.18.2 Envelopes: envelope

The `envelope` command draws the envelope of a family of curves.

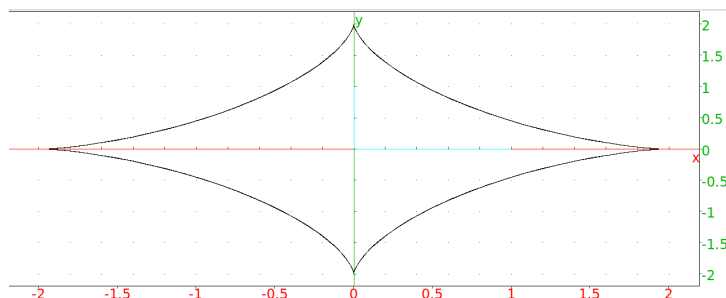
- `envelope` takes two arguments:
 - *expr*, an expression of two variables and one parameter.
 - *L*, a list of the names of the variables and the parameters. If the variables are *x* and *y*, then *L* only need to be the name of the parameter.
- `envelope(expr,L)` draws the envelope of the family of curves given by $expr = 0$ over the parameter.

Examples.

- *Input:*

```
envelope(y + x*tan(t) - 2*sin(t),t)
```

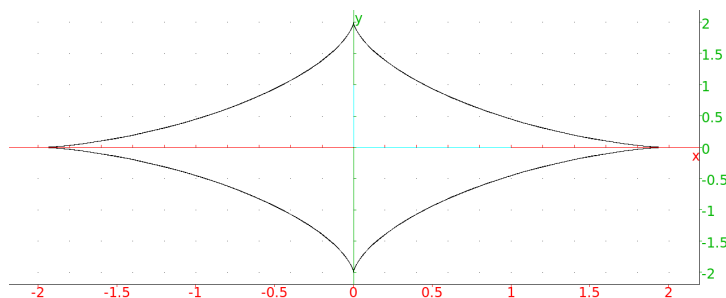
Output:



- *Input:*

```
envelope(v + u*tan(s) - 2*sin(s),[u,v,s])
```

Output:



13.18.3 The trace of a geometric object: trace

The **trace** command draws the trace of an object.

- **trace** takes one argument:
 G , a geometric object which depends on a parameter.
- **trace(G)** draws the trace of G as the parameter is changed or the object is moved in **Pointer** mode.

Example.

For example, to find the locus of points equidistant from a line D and a point F , you can create a point H on the line D . To do this, open a graphic window (**Alt-G**) and type in the following commands, one per line.

First, create a line D (using sample points) and a sample point F .

Input:

```
A:= point(-3-i)
B:= point(1/2 + 2*i)
D:= line(A,B,color=0)
F:= point(4/3,1/2,color=0)
```

Then create a point H on the line D which you can move around.

Input:

```
assume(a=[0.7, -5, 5, 0.1])
H:= element(D,a)
```

To find a point equidistant from D and F , find the point M where the perpendicular to D (at H) intersects the perpendicular bisector to HF , and trace that point.

Input:

```
T:= perpendicular(H,D)
M:= single_inter(perpen_bisector(H,F),T)
trace(M)
```

Then as the point H on the line moves (by changing the value of a with the slider), you will get the trace of M .

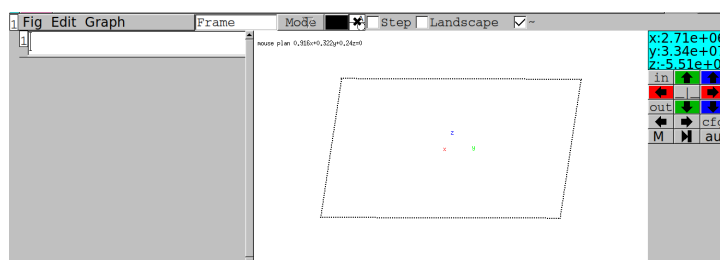
To erase traces, add traces, activate or deactivate them, use the **Trace** menu of the **M** button located on the right side of the geometry screen.

Chapter 14

Three-dimensional Graphics

14.1 Introduction

The **Alt+H** command brings up a display screen for three-dimensional graphics. This screen has its own menu and command lines.



This screen also automatically appears whenever there is a three-dimensional graphic command.

The plane of vision for a three-dimensional graphic screen is perpendicular to the observer's line of vision. The plane of vision is also indicated by dotted lines showing its intersection with the parallelepiped. The axis of vision for a three-dimensional graphic screen is

The three-dimensional graphic screen starts with the image of a parallelepiped bounding the graphics and vectors in the x , y and z directions. At the top of the screen is the equation of the plane of vision, which is a plane perpendicular to the observer's line of vision. The plane of vision is shown graphically with dotted lines indicating where it intersects the plane of vision.

Clicking in the graphic screen outside of the parallelepiped and dragging the mouse moves the x , y and z directions relative to the observer; these directions are also changed with the **x**, **X**, **y**, **Y**, **z** and **Z** keys. Scrolling the mouse wheel moves the plane of vision along the line of vision. The **in** and **out** buttons on the graphic screen menu zoom in and out of the picture.

The graphical features available for two-dimensional graphics (see Section 13.3 p.876) are also available for three-dimensional graphics, but to see the points the markers must be squares with width (**point_width**) at least 3.

The graphic screen menu has a **cfg** button which brings up a configuration screen. Among other things, this screen has

- An `Ortho proj` button, which determines whether the drawing uses orthogonal projection or perspective projection.
- A `Lights` button, which determines whether the objects are lit or not; the locations of eight points for lighting are set using the buttons `L1`, `...`, `L7`, which specify the points with homogeneous coordinates.
- A `Show axis` button, which determines whether or not the outlining parallelepiped is visible.

14.2 Changing the view

The depictions of three-dimensional objects are made with a coordinate system $Oxyz$, where the x axis is horizontal and directed right, the y axis is vertical and directed up, and the z axis is perpendicular to the screen and directed out of the screen. The depictions can be transformed by changing to a different coordinate system by setting a quaternion (see Section 13.3.2 p.879).

14.3 The axes

14.3.1 Drawing unit vectors: `0x_3d_unit_vector` `0y_3d_unit_vector` `0z_3d_unit_vector` `frame_3d`

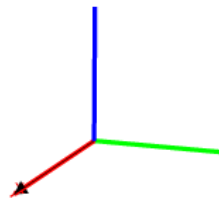
The `0x_3d_unit_vector` command takes no arguments and draws the unit vector in the x -direction on a three-dimensional graphic screen.

Example.

Input:

```
0x_3d_unit_vector()
```

Output:



Similarly, the `0y_3d_unit_vector` and `0z_3d_unit_vector` commands draw the unit vector in the y and z directions, respectively.

These commands have no parameters, but can be decorated with the `legend` command.

Example.

Input:

```
0x_3d_unit_vector(), legend(point([1,0,0]),"unit x
vector",blue)
```

Output:



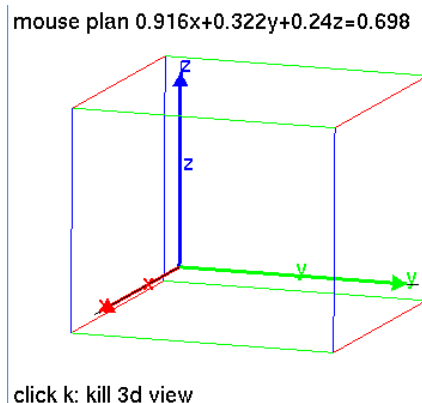
The `frame_3d` command draws all three vectors simultaneously.

Example.

Input:

```
frame_3d()
```

Output: *Output:*



14.4 Points in space

14.4.1 Defining a point in three-dimensions: `point`

See Section 13.6.2 p.886 for points in the plane.

With the 3-d geometry screen in point mode, clicking on a point with the left mouse button will choose that point. Points chosen this way are automatically named, first with A, then B, etc.

Alternatively, the `point` command chooses a point.

- `point` takes one or three arguments:
`coords`, where `coords` can be one of:
 - a, b, c , a sequence of three coordinates.

– $[a, b, c]$, a list of three coordinates.

- `point(coords)` returns and draws the point with the given coordinates.

Many commands which takes points as arguments can either take them as `point(a,b,c)` or the list of coordinates $[a, b, c]$.

Example.

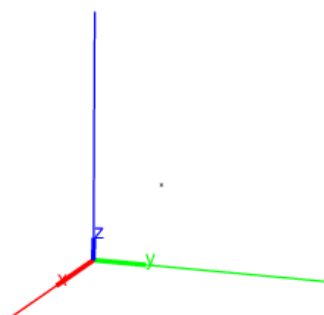
Input:

```
point(1,2,5)
```

or:

```
point([1,2,5])
```

Output:



(The marker used to indicate the point can be changed; see Section 13.3.2 p.877.)

14.4.2 Defining a random point in three-dimensions: `point3d`

The `point3d` command defines a random point whose coordinates are integers between -5 and 5.

- `point3d` takes an unspecified number of arguments:
 $names$, a sequence of names for the points.
- `point3d(names)` assigns a random point whose coordinates are integers between -5 and 5 to each name.

Example.

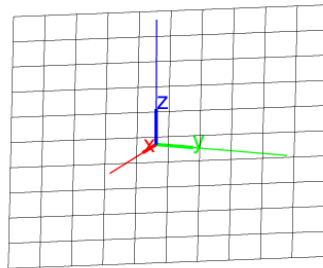
Input:

```
point3d(A,B,C)
```

then:

```
plane(A,B,C)
```

Output:



14.4.3 Finding an intersection point of two objects in space: single_inter line_inter

See Section 13.6.6 p.891 for single points of intersection of objects in the plane.

The `single_inter` command finds an intersection point of two geometric objects.

`line_inter` is a synonym for `single_inter`

- `single_inter` takes two mandatory arguments and one optional argument.
 - obj_1 , obj_2 , two geometric objects.
 - Optionally, pt , a point or list of points.

`line_inter(obj1,obj2⟨ pt ⟩)` returns one of the points of intersection of obj_1 and obj_2 .

If pt is a single point, then the command returns the point of intersection *closest* to pt .

If pt is a list of points, then the command tries to return a point not in pt .

Examples.

- *Input:*

```
A:=single_inter(plane(point(0,1,1),point(1,0,1),point(1,1,0))
               ,line(point(0,0,0),point(1,1,1)))::
coordinates(A)
```

Output:

$$\begin{bmatrix} 2 & 2 & 2 \\ 3 & 3 & 3 \end{bmatrix}$$

- *Input:*

```
B:= single_inter(sphere(point(0,0,0),1),
                 line(point(0,0,0),point(1,1,1)))::
coordinates(B)
```

Output:

$$\left[\frac{\sqrt{3}}{3}, \frac{\sqrt{3}}{3}, \frac{\sqrt{3}}{3} \right]$$

- *Input:*

```
B1:=single_inter(sphere(point(0,0,0),1),line(point(0,0,0),point(1,1,1)),
                 point(-1,0,0))::
coordinates(B1)
```

Output:

$$\left[-\frac{\sqrt{3}}{3}, -\frac{\sqrt{3}}{3}, -\frac{\sqrt{3}}{3} \right]$$

- *Input:*

```
C:= single_inter(sphere(point(0,0,0),1),line(point(1,0,0),point(1,1,1)))::
coordinates(C)
```

Output:

$$[1, 0, 0]$$

- *Input:*

```
C1:=single_inter(sphere(point(0,0,0),1),line(point(1,0,0),point(1,1,1)),
                 [point(1,0,0)])::
coordinates(C1)
```

Output:

$$\left[\frac{1}{3}, \frac{2}{3}, \frac{2}{3} \right]$$

14.4.4 Finding the intersection points of two objects in space: `inter`

See Section 13.6.7 p.892 for points of intersection of objects in the plane.

The `inter` command finds the intersection of two geometric objects in \mathbb{R}^3 .

- `inter` takes two mandatory arguments and one optional argument.
 - `obj1`, `obj2`, two geometric objects.

- Optionally, P , a point.

`inter(obj1, obj2 ⟨ P ⟩)` returns a list of points of intersection of obj_1 and obj_2 or the curve of intersection of the two objects.

With the argument P , the command returns the point of intersection *closest* to P .

Examples.

- *Input:*

```
LA:=inter(plane(point(0,1,1),point(1,0,1),point(1,1,0)),line(point(0,0,0),point(1,1,1)),
coordinates(LA)
```

Output:

$$\left[\left[\frac{2}{3}, \frac{2}{3}, \frac{2}{3} \right] \right]$$

- *Input:*

```
LB:=inter(sphere(point(0,0,0),1),line(point(0,0,0),point(1,1,1)))::
coordinates(LB)
```

Output:

$$\left[\left[\frac{\sqrt{3}}{3}, \frac{\sqrt{3}}{3}, \frac{\sqrt{3}}{3} \right], \left[-\frac{\sqrt{3}}{3}, -\frac{\sqrt{3}}{3}, -\frac{\sqrt{3}}{3} \right] \right]$$

To get just one of the points, use the usual list indices.

Input:

```
coordinates(LB[0])
```

Output:

$$\left[\frac{\sqrt{3}}{3}, \frac{\sqrt{3}}{3}, \frac{\sqrt{3}}{3} \right]$$

To get the point closest to $(1/2, 1/2, 1/2)$:

Input:

```
LB1:=inter(sphere(point(0,0,0),1),line(point(0,0,0),point(1,1,1)),point(1/2,1/2,1/2),
coordinates(LB1)
```

Output:

$$\left[\frac{\sqrt{3}}{3}, \frac{\sqrt{3}}{3}, \frac{\sqrt{3}}{3} \right]$$

14.4.5 Finding the midpoint of a segment in space: midpoint

See Section 13.6.9 p.894 for midpoints in the plane.

The `midpoint` command finds the midpoint of two points.

- `midpoint` takes two arguments:
 P, Q , two points (which can also be given as a list).
- `midpoint(P, Q)` draws and returns the midpoint of the segment determined by these points.

Example.

Input:

```
MP:= midpoint(point(1,4,0),point(1,-2,0));
coordinates(MP)
```

Output:

[1, 1, 0]

14.4.6 Finding the barycenter of a set of points in space: barycenter

See Section 13.6.10 p.894 for barycenters of objects in the plane.

The `barycenter` command returns and draws the barycenter of a set of weighted points.

- `barycenter` takes an unspecified number of arguments:
 L_1, L_2, \dots, L_n , a sequence of lists of length two, where each list consists of a point and a weight. This information can also be given as a matrix with two columns (the first column the points and the second column the weights) or a matrix with two rows and more than two columns.
- `barycenter(L1, L2, ..., Ln)` draws and returns the barycenter of the weighted points.

If the sum of the weights is zero, then this command returns an error.

Examples.

- *Input:*

```
BC:= barycenter([point(1,4,0),1],[point(1,-2,0),1])
```

or:

```
BC:= barycenter([[point(1,4,0),1],[point(1,-2,0),1]])
```

then:

```
coordinates(BC)
```

Output:

[1, 1, 0]

14.4.7 Finding the isobarycenter of a set of points in space: isobarycenter

See Section 13.6.11 p.895 for isobarycenters of objects in the plane.

The `isobarycenter` command finds the isobarycenter of a list of points; the isobarycenter is the barycenter when all points are equally weighted.

- `isobarycenter` takes one argument:
 L , a list of points. (The points can also be given by a sequence).
- `isobarycenter(L)` draws and returns the isobarycenter of the points.

Example.

Input:

```
IB:= isobarycenter(point(1,4,0),point(1,-2,0));
coordinates(IB)
```

Output:

$[1, 1, 0]$

14.5 Lines in space

14.5.1 Lines and directed lines in space: line

See Section 13.7.1 p.898 for lines in the plane.

The `line` command returns and draws a directed line. It can take its arguments in different ways.

Two points:

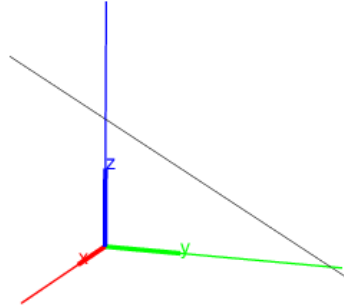
- `line` can take two arguments:
 P, Q , two points (which can also be given as a list).
- `line(P, Q)` returns and draws the line whose direction is from the P to Q .

Example.

Input:

```
line([0,3,0],point(3,0,3))
```

Output:



A point and a direction vector.

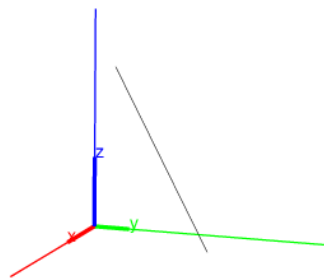
- `line` can take two arguments:
 - P , a point.
 - $[u_1, u_2, u_3]$, a direction vector.
- `line($P, [u_1, u_2, u_3]$)` returns and draws the line through the given point with the direction given by the direction vector.

Example.

Input:

```
line([0,3,0],[3,0,3])
```

Output:



Two planes.

- `line` can take two arguments:
 - eqn_1, eqn_2 , the equations of two planes.
- `line(eqn_1, eqn_2)` returns and draws the line which is the intersection of the planes.

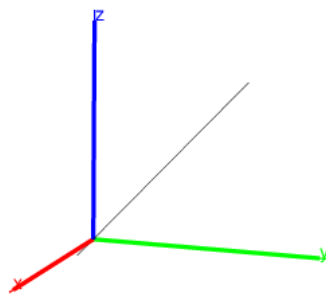
The direction of this line is given by the cross-product of the normals for the planes. For example, the intersection of the planes $x = y$ (normal $(1, -1, 0)$) and $y = z$ (normal $(0, 1, -1)$) will be $(1, -1, 0) \times (0, 1, -1) = (1, 1, 1)$.

Example.

Input:

```
line(x=y, y=z)
```

Output:



14.5.2 Half lines in space: half_line

See Section 13.7.2 p.900 for half-lines in the plane.

The `half_line` command finds rays.

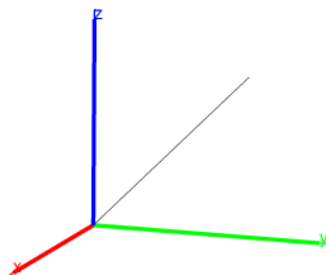
- `half_line` take two arguments:
 P, Q , two points (which can also be given as a list).
- `half_line(P, Q)` returns and draws the ray from P through Q

Example.

Input:

```
half_line(point(0,0,0),point(1,1,1))
```

Output:



14.5.3 Segments in space: `segment`

See Section 13.7.3 p.900 for segments in the plane.

The `segment` command draws line segments.

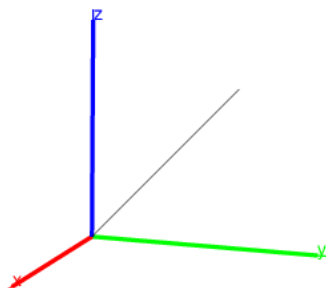
- `segment` takes two arguments:
 P, Q , two points (which can also be given as a list).
- `segment(P, Q)` returns the corresponding line segment and draws it.

Example.

Input:

```
segment(point(0,0,0),point(1,1,1))
```

Output:



14.5.4 Vectors in space: `vector`

See Section 13.7.4 p.901 for vectors in the plane.

The `vector` command returns and draws vectors. It can take its arguments in different ways.

The coordinates of the vector.

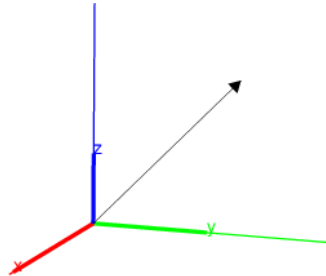
- `vector` takes one argument:
 L , a list of the coordinates of the vector.
- `vector(L)` returns and draws the vector with the given coordinates, starting from the origin.

Example.

Input:

```
vector([1,2,3])
```

Output:



Two points or a point and a vector.

- **vector** takes two arguments:
 - P , a point.
 - Q , a point or a vector.
- **vector**(P, Q) returns and draws the corresponding vector. If the arguments are two points, the vector goes from P to Q . If the arguments are a point and a vector, then the vector starts at P .

Examples.

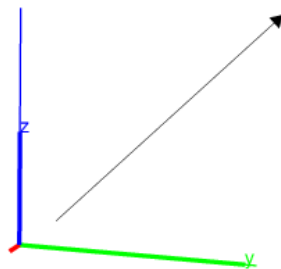
- *Input:*

```
vector(point(-1,0,0),point(0,1,2))
```

or:

```
vector([-1,0,0],[0,1,2])
```

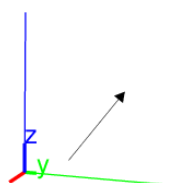
Output:



- *Input:*

```
V:= vector([-1,0,0],[0,1,2])
vector(point(-1,2,0),V)
```

Output:



14.5.5 Parallel lines and planes in space: parallel

See Section 13.7.5 p.903 for parallel lines in the plane.

The **parallel** command can take its arguments in different ways. It returns and draws a line or plane depending on the arguments.

A point and a line.

- **parallel** takes two arguments:

- P , a point.
- L , a line.

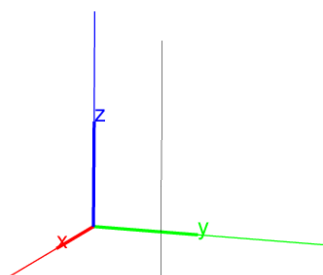
- **parallel**(P, L) returns and draws the line through P parallel to L .

Example.

Input:

```
parallel(point(1,1,1),line(point(0,0,0),point(0,0,1)))
```

Output:



Two non-parallel lines.

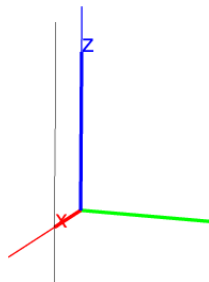
- `parallel` takes two arguments:
 L, M , two lines which aren't parallel.
- `parallel(L, M)` returns and draws the plane containing L which is parallel to M .

Example.

Input:

```
parallel(line(point(1,0,0),point(0,1,0)),line(point(0,0,0),point(0,0,1)))
```

Output:



A point and a plane.

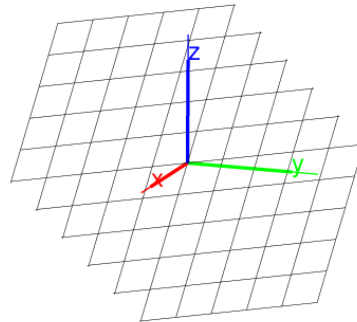
- `parallel` takes two arguments:
 - P , a point.
 - PL , a plane.
- `parallel(P, PL)` returns and draws the plane through P that is parallel to PL .

Example.

Input:

```
parallel(point(0,0,0),plane(point(1,0,0),point(0,1,0),point(0,0,1)))
```

Output:



A point and two non-parallel lines.

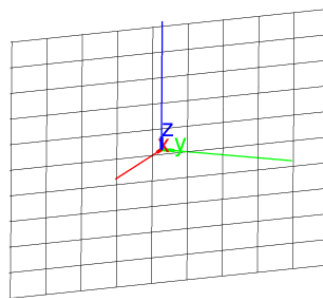
- `parallel` takes three arguments:
 - P , a point.
 - L, M , two non-parallel lines.
- `parallel(P, L, M)` returns and draws the plane through P that is parallel to L and M .

Example.

Input:

```
parallel(point(1,1,1),line(point(0,0,0),point(0,0,1)),line(point(1,0,0),point(0,0,0)))
```

Output:



14.5.6 Perpendicular lines and planes in space: perpendicular

See Section 13.7.6 p.904 for perpendicular lines in the plane.

The `perpendicular` command can take its arguments in different ways. It returns and draws a line or plane, depending on the arguments.

A point and a line.

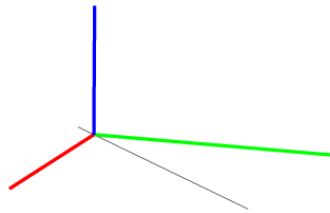
- `perpendicular` takes two arguments:
 - P , a point.
 - L , a line.
- `perpendicular(P, L)` returns and draws the line through P that is perpendicular to L .

Example.

Input:

```
perpendicular(point(0,0,0),line(point(1,0,0),point(0,1,0)))
```

Output:



A line and a plane.

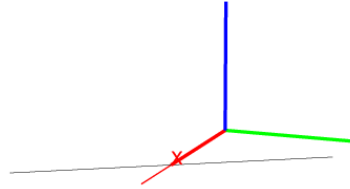
- `perpendicular` takes two arguments:
 - L , a line.
 - P , a plane.
- `perpendicular(L, P)` returns and draws the plane containing L that is perpendicular to P .

Example.

Input:

```
perpendicular(line(point(0,0,0),point(1,1,0)),plane(point(1,0,0),
point(0,1,0),point(0,0,1)))
```

Output:



14.5.7 Planes orthogonal to lines and lines orthogonal to planes in space: orthogonal

The `orthogonal` command finds orthogonal objects. It takes its arguments in different ways, and returns and draws a line or plane, depending on the arguments.

A point and a line.

- `orthogonal` takes two arguments:

- P , a point.
- L , a line.

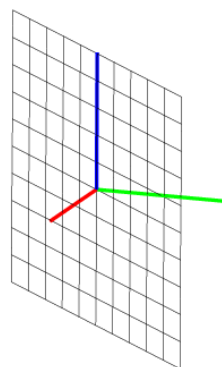
- `orthogonal(P, L)` returns and draws the plane through P orthogonal to L .

Example.

Input:

```
orthogonal(point(0,0,0),line(point(1,0,0),point(0,1,0)))
```

Output:



A line and a plane.

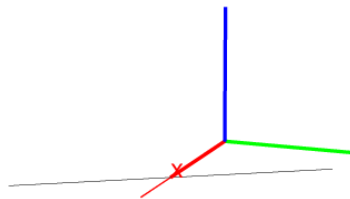
- `orthogonal` takes two arguments:
 - L , a line.
 - P , a plane.
- `orthogonal(L, P)` returns and draws the plane containing L that is perpendicular to P .

Example.

Input:

```
perpendicular(line(point(0,0,0),point(1,1,0)),
              plane(point(1,0,0),point(0,1,0),point(0,0,1)))
```

Output:



14.5.8 Common perpendiculars to lines in space: `common_perpendicular`

The `common_perpendicular` command finds the common perpendicular to two lines.

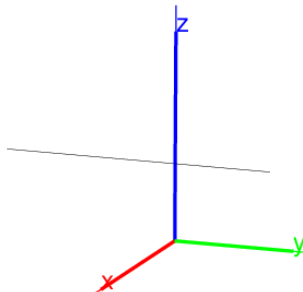
- `common_perpendicular` takes two arguments:
 - L, M , two lines.
- `common_perpendicular(L, M)` returns and draws the common perpendicular to L and M .

Example.

Input:

```
L1:= line(point(1,1,0),point(0,1,1));
L2:= line(point(0,-1,0),point(1,-1,1));
common_perpendicular(L1,L2)
```

Output:



14.6 Planes in space

See also sections 14.5.6 and 14.5.7 for planes perpendicular and orthogonal to lines and planes.

14.6.1 Planes in space: plane

The `plane` command draws and returns a plane. It can take its arguments in different ways.

- `plane` can take three arguments:
 P, Q, R , three points.
- `plane(P, Q, R)` returns and draws the plane through P, Q and R .
- `plane` can take two arguments:
 - P , a point.
 - L , a line.
- `plane(P, L)` returns and draws the plane through P and L .
- `plane` can take one argument:
 eqn , the equation of a plane.
- `plane(eqn)` returns and draws the plane with the given equation.

Example.

Input:

```
plane(point(0,0,5),point(0,5,0),point(0,0,5))
```

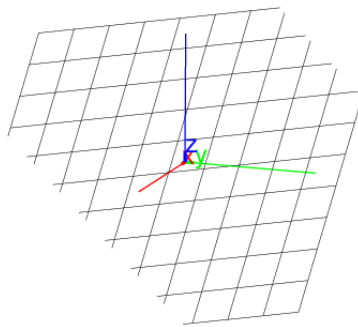
or:

```
plane(point(0,0,5),line(point(0,5,0),point(0,0,5)))
```

or:

```
plane(x + y + z = 5)
```

Output:



14.6.2 The bisector plane in space: `perpen_bisector`

See Section 13.7.10 p.906 for perpendicular bisectors in the plane.

The `perpen_bisector` command finds the perpendicular bisector plane of a line segment.

- `perpen_bisector` takes one argument:
seg, a line segment (or the end points of the segment).
- `perpen_bisector(seg)` returns and draws the perpendicular bisector plane of *seg*.

Example.

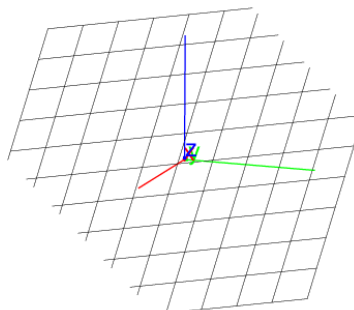
Input:

```
perpen_bisector(point(0,0,0),point(4,4,4))
```

or:

```
perpen_bisector(segment([0,0,0],[4,4,4]))
```

Output:



14.6.3 Tangent planes in space: `tangent`

See Section 13.7.7 p.904 for tangents in the plane.

The `tangent` command finds tangent planes to surfaces.

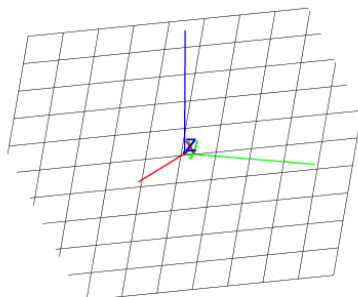
- *tangent* takes two arguments:
 - *obj*, an object in space.
 - *P*, a point in space.
 If *obj* is the graph of a function, then *P* can be a point in the domain of the function, and the point on the graph will be used.
- or
- *e*, a point defined with `element` (see Section 13.6.15 p.897) using a curve and parameter value.
- `tangent(obj P)` returns and draws the plane through *P* that's perpendicular to *obj*.

Examples.

- *Input:*

```
S: = sphere([0,0,0],3)
tangent(S,[2,2,1])
```

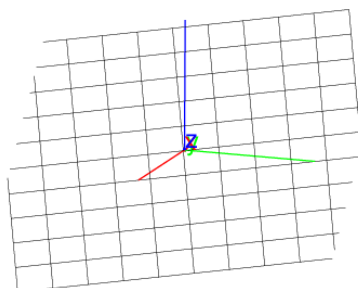
Output:



- *Input:*

```
G:=plotfunc(x^2 + y^2, [x,y])
tangent(G,[2,2])
```

Output:



14.7 Triangles in space

14.7.1 Drawing triangles in space: triangle

See Section 13.8.1 p.908 for the **triangle** command in the plane.

The **triangle** command creates triangles.

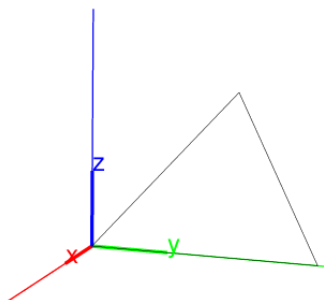
- **triangle** takes three arguments:
 A, B, C , three points.
- **triangle**(P, Q, R) returns and draws the triangle with vertices A, B and C .

Example.

Input:

```
A:= point(0,0,0); B:= point(3,3,3); C:= point(0,3,0)
triangle(A,B,C)
```

Output:



14.7.2 Isosceles triangles in space: `isosceles_triangle`

See Section 13.8.2 p.909 for isosceles triangles in the plane.

The `isosceles_triangle` command returns and draws an isosceles triangle. It can take its arguments in different ways.

Three points.

- `isosceles_triangle` takes three mandatory arguments and one optional argument:

- A, B, P , three points.
- Optionally, var , a variable name.

- `isosceles_triangle($A, B, P \langle, var \rangle$)` returns and draws the isosceles triangle ABC in the plane ABP , oriented so that angle BAC is positive and the equal interior angles of the isosceles triangle are determined by angle ABP .

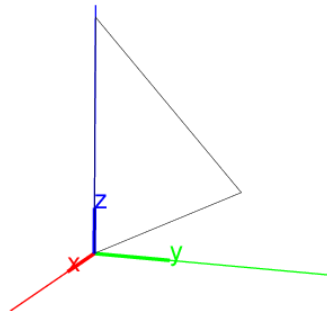
If the variable name var is given, it will be given the value of C , the third vertex of the triangle.

Example.

Input:

```
A:= point(0,0,0); B:= point(3,3,3); P:= point(0,0,3)
isosceles_triangle(A,B,P);
```

Output:



Three points and a real number.

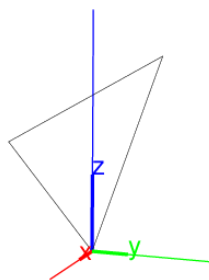
- `isosceles_triangle` takes three mandatory arguments and one optional argument:
 - A, B , two points.
 - $[P, c]$, a list consisting of a point P and a real number c .
 - Optionally, var , a variable name.
- `isosceles_triangle($A, B, [P, c], \langle var \rangle$)` returns and draws the triangle ABC in plane ABP , oriented so that angle BAC is positive. The measure of the equal interior angles is c .
If the variable name var is given, it will be given the value of the third vertex of the triangle.

Examples.

- *Input:*

```
A:= point(0,0,0); B:= point(3,3,3); P:= point(0,0,3)
isosceles_triangle(A,B,[P,3*pi/4])
```

Output:



- *Input:*

```
A:= point(0,0,0); B:= point(3,3,3); P:= point(0,0,3)
isosceles_triangle(A,B,[P,3*pi/4],C)
coordinates(C)
```

Output:

$$\left[\frac{-3\sqrt{2}-3}{2}, \frac{-3\sqrt{2}-3}{2}, \frac{-3\sqrt{2}+6}{2} \right]$$

14.7.3 Right triangles in space: right_triangle

See Section 13.8.3 p.910 for right triangles in the plane.

The `right_triangle` command returns and draws a right triangle. It can take its arguments in different ways.

Three points.

- `right_triangle` takes three mandatory arguments and one optional argument:

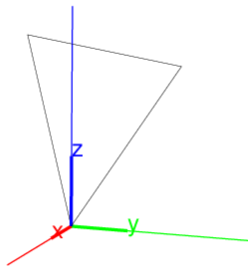
- A, B, P , three points.
- Optionally, var , a variable name.
- `right_triangle(A, B, P, ⟨var⟩)` returns and draws the right triangle BAC in plane ABP with the right angle at vertex A . The triangle is oriented so that the angle BAC is positive. The length of AC equals the length of AP .
If the variable name var is given, it will be assigned to the vertex C .

Example.

Input:

```
A:= point(0,0,0); B:= point(3,3,3);
P:= point(0,0,3)
right_triangle(A,B,P);
```

Output:



Three points and a real number.

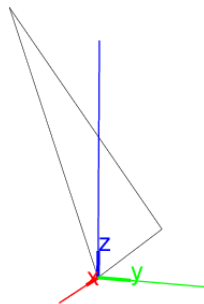
- `right_triangle` takes mandatory three arguments and one optional argument:
 - * A, B , two points.
 - * $[P, k]$, a list consisting of a point P and a real number k .
 - * Optionally, var , a variable name.
- `right_triangle($A, B, [P, k]$ $\langle, var \rangle$)` returns and draws the right triangle BAC in plane ABP with the right angle at vertex A , and the length of AC equals $|k|$ times the length of AP . Angles BAC and BAP have the same orientation if k is positive; they have opposite orientation if k is negative. So, if β is the angle ABC , then $\tan(\beta) = k$.
If the variable name var is given, it will be assigned to the vertex C .

Examples.

– *Input:*

```
right_triangle(A,B, [P,2])
```

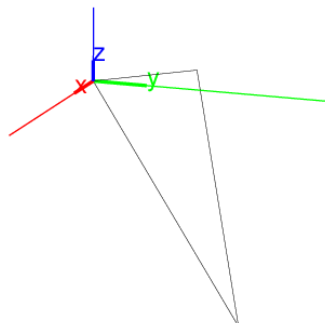
Output:



– *Input:*

```
right_triangle(A,B, [P,-2])
```

Output:



– *Input:*

```
right_triangle(A,B,[P,2],C)
coordinates(C)
```

Output:

$$\left[-3\sqrt{2}, -3\sqrt{2}, 6\sqrt{2}\right]$$

14.7.4 Equilateral triangles in space: equilateral_triangle

See Section 13.8.4 p.911 for equilateral triangles in the plane.

The `equilateral_triangle` command returns and draws equilateral triangles.

– `equilateral_triangle` takes three mandatory arguments and one optional argument:

- * A, B, P , three points.
- * Optionally, var , a variable name.

– `equilateral_triangle(A, B, P [, var])` returns and draws equilateral triangle ABC , where C and P are on the same side of line AB in plane ABP .

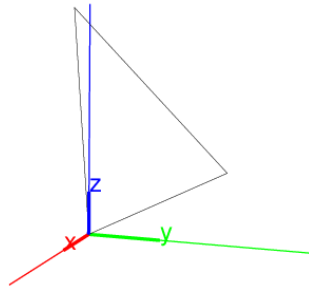
If the argument var is given, it will be assigned the value of C .

Examples.

– *Input:*

```
A:= point(0,0,0);
B:= point(3,3,3);
P:= point(0,0,3)
equilateral_triangle(A,B,P)
```

Output:



– *Input:*

```
A:= point(0,0,0);
B:= point(3,3,3);
P:= point(0,0,3)
equilateral_triangle(A,B,P,C)
simplify(coordinates(C))
```

Output:

$$\left[\frac{-3\sqrt{6} + 6}{4}, \frac{-3\sqrt{6} + 6}{4}, \frac{3\sqrt{6} + 3}{2} \right]$$

14.8 Quadrilaterals in space

See Section 13.9 p.912 for quadrilaterals in the plane.

14.8.1 Squares in space: square

See Section 13.9.1 p.913 for squares in the plane.

The **square** command creates squares.

- **square** takes three mandatory arguments and two optional arguments:
 - * A, B, P , three points.
 - * Optionally, $var1, var2$, two variable names.
- **square**($A, B, P \langle, var1, var2 \rangle$) returns and draws the square with one side AB and the remaining sides in the same half-plane as P . If the arguments $var1$ and $var2$ are given, they will be assigned to the new vertices.

Examples.

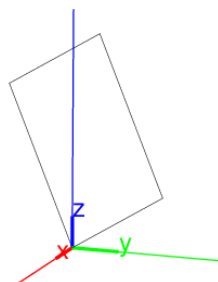
– *Input:*

```

A:= point(0,0,0);
B:= point(3,3,3);
P:= point(0,0,3);
square(A,B,P)

```

Output:



– *Input:*

```

A:= point(0,0,0); B:= point(3,3,3); P:= point(0,0,3); square(A,B,P,C,D)

```

Output:

$$\left[\frac{-3\sqrt{2}+6}{2}, \frac{-3\sqrt{2}+6}{2}, 3\sqrt{2}+3 \right], \left[-\frac{3}{2}\sqrt{2}, -\frac{3}{2}\sqrt{2}, 3\sqrt{2} \right]$$

14.8.2 Rhombuses in space: rhombus

See Section 13.9.2 p.914 for rhombuses in the plane.

The **rhombus** command returns and draws a rhombus. It takes its arguments in different ways.

Three points:

- **rhombus** takes three mandatory arguments and two optional arguments:
 - * A, B, P , three points.
 - * Optionally, $var1, var2$, two variable names.
- **rhombus**($A, B, P \langle var1, var2 \rangle$) returns and draws the rhombus $ABCD$, which is in the plane ABP , oriented so that angle BAP is positive, and D is on the ray AP .
If the arguments $var1$ and $var2$ are given, they will be assigned to the vertices C and D .

Example.

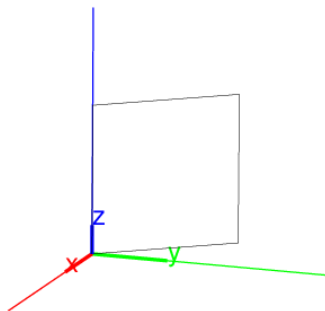
Input:

```

A:= point(0,0,0);
B:= point(3,3,3);
P:= point(0,0,3)
rhombus(A,B,P)

```

Output:



Three points and a real number.

- **rhombus** takes three mandatory arguments and two optional argument:
 - * A, B , two points.
 - * $[P, a]$, a list consisting of a point P and a real number a .
 - * Optionally, $var1, var2$, two variable names.
- **rhombus**($A, B, [P, a] \langle var1, var2 \rangle$) returns and draws the rhombus $ABCD$, which is in the plane ABP , oriented so that angle BAP is positive, and angle BAD equals a .
If the arguments $var1$ and $var2$ are given, they will be assigned to the vertices C and D .

Examples.

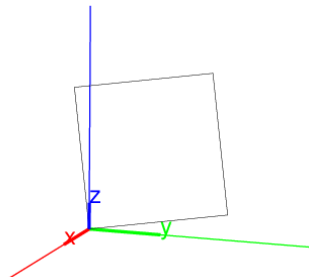
– *Input:*

```

A:= point(0,0,0);
B:= point(3,3,3);
P:= point(0,0,3)
rhombus(A,B,[P,pi/3])

```

Output:



– *Input:*

```
rhombus(A,B,[P,pi/3],C,D)
simplify(coordinates(C)), simplify(coordinates(D))
```

Output:

$$\left[\frac{-3\sqrt{6}+18}{4}, \frac{-3\sqrt{6}+18}{4}, \frac{3\sqrt{6}+9}{2} \right], \left[\frac{-3\sqrt{6}+6}{4}, \frac{-3\sqrt{6}+6}{4}, \frac{3\sqrt{6}+3}{2} \right]$$

14.8.3 Rectangles in space: rectangle

See Section 13.9.3 p.915 for rectangles in the plane.

The **rectangle** command returns and draws a rectangle. It can take its arguments in different ways.

Three points.

– **rectangle** takes three mandatory arguments and two optional arguments:

- * A, B, P , three points.
- * Optionally, $var1, var2$, two variable names.

– **rectangle**($A, B, P \langle var1, var2 \rangle$) returns and draws the rectangle $ABCD$, in the plane ABP , oriented to that angle BAP is positive, and with the length of side AD equals AP .

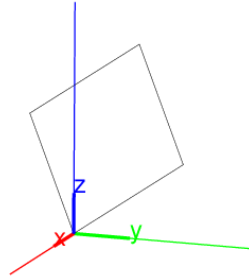
If the arguments $var1$ and $var2$ are given, they will be assigned to the vertices C and D .

Example.

Input:

```
A:= point(0,0,0);
B:= point(3,3,3);
P:= point(0,0,3)
rectangle(A,B,P)
```

Output:



Three points and a real number.

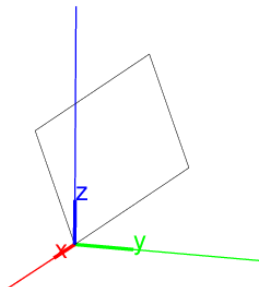
- **rectangle** takes three mandatory arguments and two optional argument:
 - * A, B , two points.
 - * $[P, k]$, a list consisting of a point P and a real number k .
 - * Optionally, $var1, var2$, two variable names.
- **rectangle**($A, B, [P, k] \langle var1, var2 \rangle$) returns and draws the rectangle $ABCD$, which is in the plane ABP , and with the length of AD equal to $|k|$ times the length of AB . Angle BAD and angle BAP have the same orientation if k is positive and opposite orientation if k is negative. If the arguments $var1$ and $var2$ are given, they will be assigned to the vertices C and D .

Examples.

– *Input:*

```
A:= point(0,0,0);
B:= point(3,3,3);
P:= point(0,0,3)
rectangle(A,B, [P,1/2])
```

Output:



– *Input:*

```
rectangle(A,B,P,C,D)
simplify(coordinates(C)), simplify(coordinates(D))
```

Output:

$$\left[-\frac{\sqrt{6}}{2}, -\frac{\sqrt{6}}{2}, \sqrt{6} \right], \left[\frac{-\sqrt{6}+6}{2}, \frac{-\sqrt{6}+6}{2}, \sqrt{6}+3 \right]$$

14.8.4 Parallelograms in space: parallelogram

See Section 13.9.4 p.916 for parallelograms in the plane.

The `parallelogram` command creates parallelograms in space.

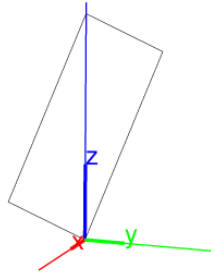
- `parallelogram` takes three mandatory arguments and one optional argument:
 - * A, B, C , three points.
 - * var , a variable name.
- `parallelogram(A,B,C⟨,var⟩)` returns and draws the parallelogram $ABCD$ determined by A, B and C .
If the option var is given, the point D will be assigned to it.

Examples.

– *Input*

```
A:= point(0,0,0);
B:= point(3,3,3);
C:= point(0,0,3)
parallelogram(A,B,C)
```

Output:



– *Input:*

```
parallelogram(A,B,C,D)
coordinates(D)
```

Output:

$[-3, -3, 0]$

14.8.5 Arbitrary quadrilaterals in space: quadrilateral

See Section 13.9.5 p.918 for quadrilaterals in the plane.

The `quadrilateral` command creates arbitrary quadrilaterals.

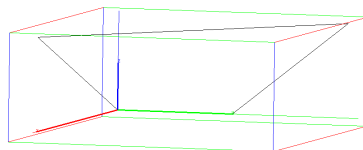
- `quadrilateral` takes four arguments:
 A, B, C, D , four points.
- `quadrilateral(A, B, C, D)` returns and draws quadrilateral $ABCD$.

Example.

Input:

```
quadrilateral(point(0,0,0),point(0,1,0),point(0,2,2),point(1,0,2))
```

Output:



14.9 Polygons in space

See Section 13.10 p.918 for polygons in the plane.

14.9.1 Hexagons in space: hexagon

See Section 13.10.1 p.918 for hexagons in the plane.

The `hexagon` command creates hexagons in space.

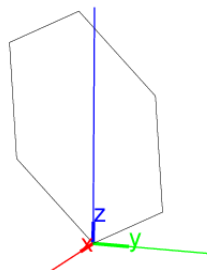
- `hexagon` takes three mandatory arguments and four optional arguments:
 - * A, B, P , four points.
 - * Optionally, $var1, var2, var3, var4$, four variable names.
- `hexagon(A, B, P ⟨,var1,var2,var3,var4⟩)` returns and draws the regular hexagon $ABCDEF$ in the plane ABP , oriented so that angle ABC is positive.

Examples.

– *Input:*

```
A:= point(0,0,0);
B:= point(3,3,3);
P:= point(0,0,3);
hexagon(A,B,P)
```

Output:



– *Input:*

```
hexagon(A,B,P,C,D,E,F)
simplify(coordinates(C))
```

Output:

$$\left[\frac{-3\sqrt{6} + 18}{4}, \frac{-3\sqrt{6} + 18}{4}, \frac{3\sqrt{6} + 9}{2} \right]$$

14.9.2 Regular polygons in space: isopolygon

See Section 13.10.2 p.919 for regular polygons in the plane.

The `isopolygon` command creates regular polygons in space.

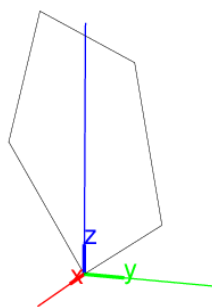
- `isopolygon` takes four arguments:
 - * A, B, P , three points.
 - * k , an integer.
- `isopolygon(A, B, P, k)` returns and draws a regular polygon with one edge AB in the plane ABP with $|k|$ sides. If $|k|$ is positive, then the polygon is positively oriented, otherwise it is negatively oriented.

Examples.

– *Input:*

```
A:= point(0,0,0);
B:= point(3,3,3);
P:= point(0,0,3);
isopolygon(A,B,P,5)
```

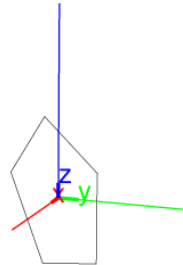
Output:



– *Input:*

```
isopolygon(A,B,P,-5)
```

Output:



14.9.3 General polygons in space: polygon

See Section 13.10.3 p.920 for general polygons in the plane.

The `polygon` command creates general polygons in space.

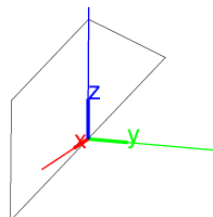
- `polygon` takes one argument:
 S , a sequence of points.
- `polygon(S)` returns and draws the polygon whose vertices are the given points.

Example.

Input:

```
A:= point(0,0,0);
B:= point(3,3,3);
C:= point(0,0,3);
D:= point(-3,-3,0);
E:= point(-3,-3,-3)
polygon(A,B,C,D,E)
```

Output:



14.9.4 Polygonal lines in space: `open_polygon`

See Section 13.10.4 p.921 for polygonal lines in the plane.

The `open_polygon` command creates polygonal lines in space.

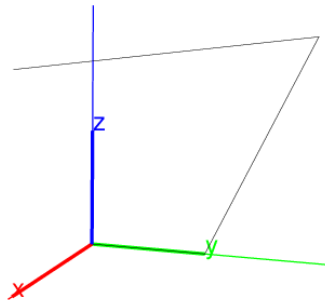
- `open_polygon` takes one argument:
 S , a sequence of points.
- `open_polygon(S)` returns and draws the polygon line whose vertices are the given points.

Example.

Input:

```
open_polygon(point(0,0,0),point(0,1,0),point(0,2,2),point(1,0,2))
```

Output:



14.10 Circles in space: `circle`

See Section 13.11.1 p.923 for circles in the plane.

The `circle` command returns and draws a circle. It can take its arguments in various ways.

Three points.

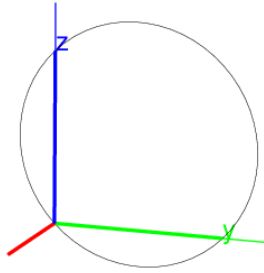
- `circle` takes three arguments:
 A, B, C , three points.
- `circle(A, B, C)` returns and draws the circle in plane ABC with a diameter AB .

Example.

Input:

```
circle(point(0,0,1),point(0,1,0),point(0,2,2))
```

Output:



Two points and a vector.

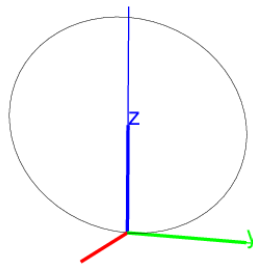
- **circle** takes three points:
 - * C , a point (which can be given by its coordinates).
 - * v , a vector.
 - * A , a point (which can be given by its coordinates).
- **circle**(C, v, A) returns and draws the circle in plane $C(C+v)C$ with center C and containing $C + v$.

Example.

Input:

```
circle(point(0,0,1),vector(0,1,0),point(0,2,2))
```

Output:



14.11 Conics in space

14.11.1 Ellipses in space: ellipse

See Section 13.12.1 p.929 for ellipses in the plane.

The `ellipse` command creates ellipses in space.

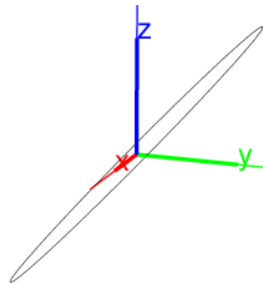
- `ellipse` takes three arguments:
 A, B, C three non-collinear points.
- `ellipse(A, B, C)` returns and draws the ellipse with foci A and B passing through C .

Example.

Input:

```
ellipse(point(-1,0,0),point(1,0,0),point(1,1,1))
```

Output:



14.11.2 Hyperbolas in space: hyperbola

See Section 13.12.2 p.930 for hyperbolas in the plane.

The `hyperbola` command creates hyperbolas in space.

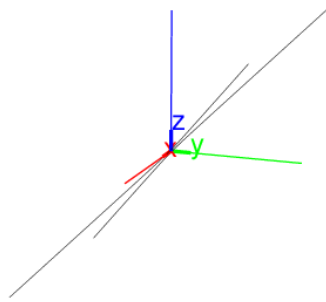
- `hyperbola` takes three arguments:
 A, B, C three non-collinear points.
- `hyperbola(A, B, C)` returns and draws the hyperbola with foci A and B passing through C .

Example.

Input:

```
hyperbola(point(-1,0,0),point(1,0,0),point(1,1,1))
```

Output:



14.11.3 Parabolas in space: parabola

See Section [13.12.3](#) p.932 for parabolas in the plane.

The **parabola** command creates parabolas in space.

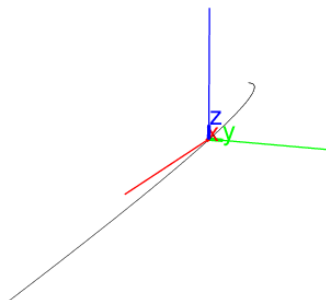
- **parabola** takes three arguments:
 A, B, C three non-collinear points.
- **parabola**(A, B, C) returns and draws the parabola in plane ABC with focus A and vertex B .

Example.

Input:

```
parabola(point(0,0,0),point(-1,0,0),point(1,1,1))
```

Output:



14.12 Three-dimensional coordinates

14.12.1 The abscissa of a three-dimensional point: `abscissa`

See Section [13.13.2](#) p.934 for abscissas in two-dimensional geometry.

The `abscissa` command finds the abscissa (x -coordinate) of a point.

- `abscissa` takes one argument:
 P , a point.
- `abscissa(P)` returns the abscissa of P .

Example.

Input:

```
abscissa(point(1,2,3))
```

Output:

1

14.12.2 The ordinate of a three-dimensional point: `ordinate`

See Section [13.13.3](#) p.935 for ordinates in two-dimensional geometry.

The `ordinate` command finds the ordinate (y -coordinate) of a point.

- `ordinate` takes one argument:
 P , a point.
- `ordinate(P)` returns the ordinate of P .

Example.

Input:

```
ordinate(point(1,2,3))
```

Output:

2

14.12.3 The cote of a three-dimensional point: `cote`

The `cote` command finds the cote (z -coordinate) of a point.

- `cote` takes one argument:
 P , a point.
- `cote(P)` returns the cote of P .

Example.

Input:

```
cote(point(1,2,3))
```

Output:

3

14.12.4 The coordinates of a point, vector or line in space: `coordinates`

See Section 13.13.4 p.935 for coordinates in two-dimensional geometry.

The `coordinates` command takes finds the coordinates of a point.

- `coordinates` takes one argument:
 P , which can be a point (or a sequence or list of points), a vector, or a line.
- If P is a point, then `coordinates(P)` returns a list consisting of the abscissa, ordinate and cote. If P is a list or sequence of points, then the command returns a list or sequence of such lists.
- If P is a vector, for example from A to B , then `coordinates(P)` returns a list of the coordinates of $B - A$.
- If P is a line, then `coordinates(P)` returns a list of two points on the line, in the order determined by the direction of the line.

Examples.

- *Input:*

```
coordinates(point(1,2,3))
```

Output:

[1, 2, 3]

- *Input:*

```
coordinates(point(0,1,2),point(1,2,4))
```

Output:

[0, 1, 2], [1, 2, 4]

- Note that if the argument is a list of real numbers, it is interpreted as a list of points on the real axis of the plane.

Input:

```
coordinates([1,2,4])
```

Output:

$$\begin{bmatrix} 1 & 0 \\ 2 & 0 \\ 4 & 0 \end{bmatrix}$$

- *Input:*

```
coordinates(vector(point(1,2,3),point(2,4,7)))
```

Output:

$$[1, 2, 4]$$

– *Input:*

```
coordinates(line(point(-1,1,0),point(1,2,3)))
```

Output:

$$[[-1, 1, 0], [1, 2, 3]]$$

– *Input:*

```
coordinates(line(x-2*y+3=0, 6*x + 3*y - 5*z + 3 = 0))
```

Output:

$$[[-1, 1, 0], [9, 6, 15]]$$

14.12.5 The Cartesian equation of an object in space: equation

See Section [13.13.7](#) p.939 for Cartesian equations of two-dimensional objects.

The `equation` command finds equations for geometric objects.

- `equation` takes one argument:
 G , a geometric object.
- `equation(G)` returns Cartesian equations in x , y and z which specify the object G .
 The variables x , y and z must be unassigned. If they have assignments, they can be unassigned with `purge(x,y,z)`.

Examples.

– *Input:*

```
equation(line(point(0,1,0),point(1,2,3)))
```

Output:

$$x - y + 1 = 0, 3x + 3y - 2z - 3 = 0$$

– *Input:*

```
equation(sphere(point(0,1,0),2))
```

Output:

$$x^2 + y^2 - 2y + z^2 - 3 = 0$$

14.12.6 The parametric equation of an object in space: `parameq`

See Section 13.13.8 p.939 for parametric equations in two-dimensional geometry.

The `parameq` command finds parameterizations for geometric objects.

- `parameq` takes one argument:
 G , a geometric object.
- `parameq(G)` returns a parameterization for the object G .
 For a curve, the parameter is t , for a surface, the parameters are u and v . These variables must be unassigned. If they have assignments, they can be unassigned with `purge(t)` and `purge(u, v)`.

Examples.

– *Input:*

```
parameq(line(point(0,1,0),point(1,2,3)))
```

Output:

$$[t, t + 1, 3t]$$

– *Input:*

```
parameq(sphere(point(0,1,0),2))
```

Output:

$$[2 \cos u \cdot \cos v, 1 + 2 \cos u \cdot \sin v, 2 \sin u]$$

– *Input:*

```
normal(parameq(ellipse(point(-1,1,1),point(1,1,1),point(0,1,2))))
```

Output:

$$\left[\sqrt{2} \cos t, 1, \sin t + 1 \right]$$

14.12.7 The length of a segment in space: `distance`

See Section 13.14.2 p.942 for distances in two-dimensional geometry.

The `distance` command finds the distance between two points.

- `distance` takes two arguments:
 P, Q , two points or two lists with the coordinates of the points.
- `distance(P, Q)` returns the distance between P and Q .

Example.

Input:

```
distance(point(-1,1,1),point(1,1,1))
```

or:

```
distance([-1,1,1],[1,1,1])
```

Output:

2

14.12.8 The length squared of a segment in space: distance2

See Section [13.14.3 p.943](#) for squares of lengths in two-dimensional geometry.

The `distance2` command finds the square of the distance between two points.

- `distance2` takes two arguments:
 P, Q , two points or two lists with the coordinates of the points.
- `distance2(P, Q)` returns the square of the distance between P and Q .

Example.

Input:

```
distance2(point(-1,1,1),point(1,1,1))
```

or:

```
distance2([-1,1,1],[1,1,1])
```

Output:

4

14.12.9 The measure of an angle in space: angle

See Section [13.14.4 p.943](#) for angle measures in two-dimensional geometry.

The `angle` command finds the measures of angles in space. It can take its arguments in different ways.

Three points.

- `angle` takes three arguments:
 A, B, C , three points.
- `angle(A, B, C)` returns the measure of the undirected angle BAC .

Example.

Input:

```
angle(point(0,0,0),point(1,0,0),point(0,0,1))
```

Output:

$$\frac{1}{2}\pi$$

Two intersecting lines.

- **angle** takes two arguments:
 L, M , two lines which intersect.
- **angle**(L, M) returns the measure of the angle between the lines L and M .

Example.

Input:

```
angle(line([0,0,0],[1,1,0]),line([0,0,0],[1,1,1]))
```

Output:

$$\arccos\left(\frac{\sqrt{6}}{3}\right)$$

A line and a plane.

- **angle** takes two arguments:
 - * L , a line.
 - * P , a plane.
- **angle**(L, P) returns the measure of the angle between L and P .

Example.

Input:

```
angle(line([0,0,0],[1,1,0]),plane(x+y+z=0))
```

Output:

$$\arccos\left(\frac{\sqrt{6}}{3}\right)$$

14.13 Properties

14.13.1 Checking if an object in space is on another object: `is_element`

See Section 13.16.1 p.960 for checking elements in two-dimensional geometry.

The `is_element` command determines whether or not a geometric object is contained in another.

- `is_element` takes two arguments:
 G, H , two geometric objects.

- `is_element(G, H)` returns 1 if G is contained in H and returns 0 otherwise.

Examples.

- *Input:*

```
P:= plane([0,0,0],[1,2,-3],[1,1,-2])
is_element(point(2,3,-5),P)
```

Output:

1

- *Input:*

```
L:= line([2,3,-2],[-1,-1,-1]);
P:= plane([-1,-1,-1],[1,2,-3],[1,1,-2]);
is_element(L,P)
```

Output:

0

14.13.2 Checking if points and/or lines in space are coplanar:**`is_coplanar`**

The `is_coplanar` command determines whether or not several points or several lines are coplanar.

- `is_coplanar` takes one argument:
 S , a sequence where each element is a point or a line.
- `is_coplanar(S)` returns 1 if the elements of S are coplanar and returns 0 otherwise.

Examples.

- *Input:*

```
is_coplanar([0,0,0],[1,2,-3],[1,1,-2],[2,1,-3])
```

Output:

1

- *Input:*

```
is_coplanar([-1,2,0],[1,2,-3],[1,1,-2],[2,1,-3])
```

Output:

0

- *Input:*

```
is_coplanar([0,0,0],[1,2,-3],line([1,1,-2],[2,1,-3]))
```

Output:

1

- *Input:*

```
is_coplanar(line([-1,2,0],[1,2,-3]),line([1,1,-2],[2,1,-3]))
```

Output:

0

14.13.3 Checking if lines and/or planes in space are parallel:

is_parallel

See Section 13.16.11 p.969 for checking for parallels in two-dimensional geometry.

The `is_parallel` command determines if two objects are parallel.

- `is_parallel` takes two arguments:
 L, P , each one either a line or a plane.
- `is_parallel(L, P)` returns 1 if L and P are parallel and returns 0 otherwise.

Examples.

- *Input:*

```
L1:= line([0,0,0],[-1,-1,-1])
L2:= line([2,3,-2],[-1,-1,-1])
is_parallel(L1,L2)
```

Output:

0

- *Input:*

```
P:= plane([-1,-1,-1],[1,2,-3],[0,0,0])
is_parallel(P,L2)
```

Output:

1

- *Input:*

```
P1:= plane([0,0,0],[1,2,-3],[1,1,-2])
P2:= plane([1,1,0],[2,3,-3],[2,2,-2])
is_parallel(P1,P2)
```

Output:

1

14.13.4 Checking if lines and/or planes in space are perpendicular: `is_perpendicular`

See Section 13.16.12 p.970 for checking for perpendicularity in two-dimensional geometry.

The `is_perpendicular` command determines if two objects are perpendicular.

- `is_perpendicular` takes two arguments:
 L, P , each one either a line or a plane.
- `is_perpendicular(L, P)` returns 1 if L and P are perpendicular and returns 0 otherwise.

Note that two lines must be coplanar to be perpendicular.

Examples.

- *Input:*

```
is_perpendicular(line([2,3,-2],[-1,-1,-1]),line([1,0,0],[1,2,8]))
```

Output:

0

- *Input:*

```
P1:= plane([0,0,0],[1,2,-3],[1,1,-2])
P2:= plane([-1,-1,-1],[1,2,-3],[0,0,0])
is_perpendicular(P1,P2)
```

Output:

1

- *Input:*

```
L:= plane([2,3,-2],[-1,-1,-1])
is_perpendicular(L,P1)
```

Output:

0

14.13.5 Checking if two lines or two spheres in space are orthogonal: `is_orthogonal`

See Section 13.16.13 p.970 for checking for orthogonality in two-dimensional geometry.

The `is_orthogonal` command determines whether or not two objects are orthogonal.

- `is_orthogonal` takes two arguments:
 L, P , which can be two lines, two spheres, two planes or a line and a plane.
- `is_orthogonal(L, P)` returns 1 if the objects are orthogonal; it returns 0 otherwise.

Examples.

- *Input:*

```
is_orthogonal(line([2,3,-2],[-1,-1,-1]),line([1,0,0],[1,2,8]))
```

Output:

1

- *Input:*

```
is_orthogonal(line([2,3,-2],[-1,-1,-1]),plane([-1,-1,-1],[-1,0,3],[-2,0,0]))
```

Output:

1

- *Input:*

```
is_orthogonal(plane([0,0,0],[1,2,-3],[1,1,-2]),
               plane([-1,-1,-1],[1,2,-3],[0,0,0]))
```

Output:

1

- *Input:*

```
is_orthogonal(sphere([0,0,0],sqrt(2)),sphere([2,0,0],sqrt(2)))
```

Output:

1

14.13.6 Checking if points in space are collinear: `is_collinear`

See Section [13.16.2](#) p.961 for checking for collinearity in two-dimensional geometry.

The `is_collinear` command determines whether or not points in space are collinear.

- `is_collinear` takes one argument:
 L , a list or sequence of points.

- `is_collinear(L)` returns 1 if the points in L are collinear, it returns 0 otherwise.

Examples.

- *Input:*

```
is_collinear([2,0,0],[0,2,0],[1,1,0])
```

Output:

1

- *Input:*

```
is_collinear([2,0,0],[0,2,0],[0,1,1])
```

Output:

0

14.13.7 Checking if points in space are concyclic: `is_concyclic`

See Section 13.16.3 p.962 for checking for concyclicity in two-dimensional geometry.

The `is_concyclic` command determines whether or not points are cyclic.

- `is_concyclic` takes one argument:
 L , a list or sequence of points.
- `is_concyclic(L)` returns 1 if the points in L all lie on the same circle, and returns 0 otherwise.

Examples.

- *Input:*

```
is_concyclic([2,0,0],[0,2,0],[sqrt(2),sqrt(2),0],  
             [0,0,2],[2/sqrt(3),2/sqrt(3),2/sqrt(3)])
```

Output:

1

- *Input:*

```
is_concyclic([2,0,0],[0,2,0],[1,1,0],[0,0,2],[1,1,1])
```

Output:

0

14.13.8 Checking if points in space are cospherical: `is_cospherical`

The `is_cospherical` command determines whether or not points are co-spherical.

- `is_cospherical` takes one argument:
 L , a list or sequence of points.
- `is_cospherical(L)` returns 1 if the points in L all lie on the same sphere, and returns 0 otherwise.

Examples.

- *Input:*

```
is_cospherical([2,0,0],[0,2,0],[sqrt(2),sqrt(2),0],
               [0,0,2],[2/sqrt(3),2/sqrt(3),2/sqrt(3)])
```

Output:

1

- *Input:*

```
is_cospherical([2,0,0],[0,2,0],[1,1,0],[0,0,2],[1,1,1])
```

Output:

0

14.13.9 Checking if an object in space is an equilateral triangle: `is_equilateral`

See Section [13.16.5 p.963](#) for checking for equilateral triangles in two-dimensional geometry.

The `is_equilateral` command determines whether or not a geometric object is an equilateral triangle.

- `is_equilateral` takes one argument:
 G , a geometric object or a sequence of three points assumed to be the vertices of a triangle.
- `is_equilateral(G)` returns 1 if the object is an equilateral triangle and returns 0 otherwise.

Examples.

- *Input:*

```
is_equilateral([2,0,0],[0,0,0],[1,sqrt(3),0])
```

Output:

1

- *Input:*

```
T:= triangle_equilateral([2,0,0],[0,0,0],[1,sqrt(3),0])
is_equilateral(T)
```

Output:

1

- *Input:*

```
is_equilateral([2,0,0],[0,2,0],[1,1,0])
```

Output:

0

14.13.10 Checking if an object in space is an isosceles triangle: `is_isosceles`

See Section [13.16.6](#) p.964 for checking for isosceles triangles in two-dimensional geometry.

The `is_isosceles` command determines whether or not a geometric object is an isosceles triangle.

- `is_isosceles` takes one argument:
 G , a geometric object or a sequence of three points assumed to be the vertices of a triangle.
- `is_isosceles(G)` returns 1, 2 or 3 if the object is an isosceles triangle (the number indicates which vertex is on two equal sides), returns 4 if the object is an equilateral triangle, and returns 0 otherwise.

Examples.

- *Input:*

```
is_isosceles([2,0,0],[0,0,0],[0,2,0])
```

Output:

2

- *Input:*

```
T:= triangle_isosceles([0,0,0],[2,2,0],[2,2,2])
is_isosceles(T)
```

Output:

1

- *Input:*

```
is_isosceles([1,1,0],[-1,1,0],[-1,0,0])
```

Output:

0

14.13.11 Checking if an object in space is a right triangle or a rectangle: `is_rectangle`

See Section 13.16.7 p.965 for checking for right triangles and rectangles in two-dimensional geometry.

The `is_rectangle` command determines whether or not a geometric object is an rectangle or a right triangle.

- `is_rectangle` takes one argument:
 G , a geometric object or a sequence of three or four points assumed to be the vertices of a triangle or a quadrilateral.
- `is_rectangle(G)` returns:
 - (for triangle G) 1, 2 or 3 if G is a right triangle (the number indicates which vertex is has the right angle).
 - (for quadrilaterals G) 1 if G is a rectangle but not a square.
 - (for quadrilaterals G) 2 if G is square.
 - 0 otherwise.

Examples.

- *Input:*

```
is_rectangle([2,0,0],[2,2,0],[0,2,0])
```

Output:

2

- *Input:*

```
is_rectangle([2,2,0],[-2,2,0],[-2,-1,0],[2,-1,0])
```

Output:

1

14.13.12 Checking if an object in space is a square: `is_square`

See Section 13.16.8 p.966 for checking for squares in two-dimensional geometry.

The `is_square` command determines whether or not a geometric object is a square.

- `is_square` takes one argument:
 G , a geometric object or a sequence of four points assumed to be the vertices of a quadrilateral.
- `is_square(G)` returns 1 if the object is a square and returns 0 otherwise.

Examples.

- *Input:*

```
is_square([2,2,0],[-2,2,0],[-2,-2,0],[2,-2,0])
```

Output:

1

- *Input:*

```
S:= square([0,0,0],[2,0,0],[0,0,1])
is_square(S)
```

Output:

1

- *Input:*

```
is_square([2,2,0],[-2,2,0],[-2,-1,0],[2,-1,0])
```

Output:

0

14.13.13 Checking if an object in space is a rhombus: is_rhombus

See Section 13.16.9 p.967 for checking for rhombuses in two-dimensional geometry.

The `is_rhombus` command determines whether or not a geometric object is a rhombus.

- `is_rhombus` takes one argument:
 G , a geometric object or a sequence of four points assumed to be the vertices of a quadrilateral.
- `is_square(G)` returns 1 if G is a rhombus but not a square, returns 2 if G is a square and returns 0 otherwise.

Examples.

- *Input:*

```
is_rhombus([2,0,0],[0,1,0],[-2,0,0],[0,-1,0])
```

Output:

1

- *Input:*

```
R:= rhombus([0,0,0],[2,0,0],[[0,0,1],pi/4])
is_rhombus(S)
```

Output:

1

- *Input:*

```
is_rhombus([2,2,0],[-2,2,0],[-2,-1,0],[2,-1,0])
```

Output:

0

14.13.14 Checking if an object in space is a parallelogram: `is_parallelogram`

See Section 13.16.10 p.968 for checking for parallelograms in two-dimensional geometry.

The `is_parallelogram` command determines whether or not an object is a parallelogram.

- `is_parallelogram` takes one argument:
 G , a geometric object or a sequence of four points assumed to be the vertices of a quadrilateral.
- `is_parallelogram(G)` returns 1 if G is a parallelogram, but not a rhombus or a rectangle, returns 2 if G is a rhombus but not a rectangle, returns 3 if G is a rectangle but not a square, returns 4 if G is a square, and returns 0 otherwise.

Examples.

- *Input:*

```
is_parallelogram([0,0,0],[2,0,0],[3,1,0],[1,1,0])
```

Output:

1

- *Input:*

```
is_parallelogram([-1,0,0],[0,1,0],[2,0,0],[0,-1,0])
```

Output:

0

- *Input:*

```
P:= parallelogram([0,0,0],[2,0,0],[1,1,0])
is_parallelogram(P)
```

Output:

1

- Note that

Input:

```
P := parallelogram([0,0,0],[2,0,0],[1,1,0],D)
```

defines **P** to be a list consisting of the parallelogram and the point **D**; to test if the object is a parallelogram, the first component of **P** needs to be tested.

Input:

```
is_parallelogram(P[0])
```

Output:

1

- *Input:*

```
is_parallelogram([-1,0,0],[0,1,0],[2,0,0],[0,-1,0])
```

Output:

0

14.14 Transformations in space

14.14.1 General remarks

The transformations in this section operate on any geometric object. They take as arguments parameters to specify the transformation. They can optionally take a geometric object as the last argument, in which case the transformed object is returned. Without the geometric object as an argument, these transformations will return a new command which performs the transformation. For example, to move an object **P** 3 units up, either

```
translation([0,0,3],P)
```

or

```
t := translation([0,0,3])
t(P)
```

works.

14.14.2 Translation in space: translation

See Section 13.15.2 p.951 for translations in the plane.

The **translation** command creates a translation.

- **translation** takes one mandatory argument and one optional argument:
 - v , the translation vector, which can be given as a vector or a list of coordinates.

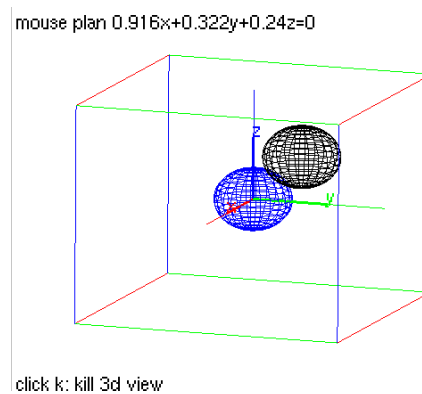
- Optionally, G , a geometric object.
- `translation(v)` returns a new command which translates by v .
- `translation(v, G)` returns and draws the translation G by the vector v .

Examples.

- *Input:*

```
t:= translation([1,1,1])
S:= sphere([0,0,0],0.5)
color(S,blue),t(S)
```

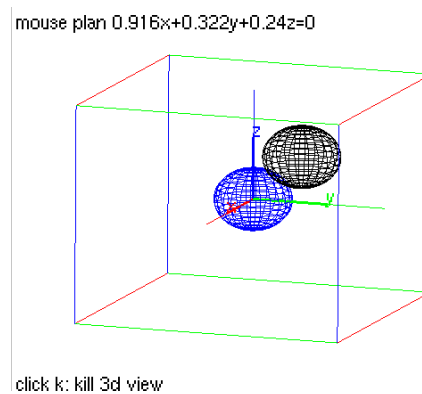
Output:



- *Input:*

```
translation([1,1,1],S)
```

Output:



14.14.3 Reflection in space with respect to a plane, line or point: reflection symmetry

See Section 13.15.3 p.952 for reflections in the plane.

The `reflection` command creates a reflection.

- `reflection` takes one mandatory argument and one optional argument:
 - P , a point, line or plane.
 - Optionally, G , a geometric object.
- `reflection(P)` returns a new command which reflects about P .
- `reflection(P, G)` returns and draws the reflection of G about P .

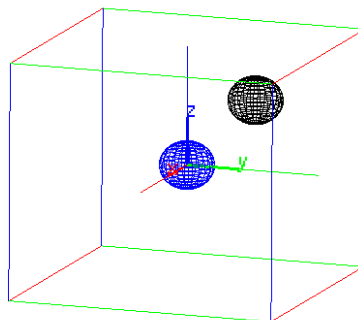
Examples.

- *Input:*

```
S:=sphere([0,0,0],0.5)
r:= reflection([1,1,1])
color(S,blue),r(S)
```

Output:

mouse plan 0.916x+0.322y+0.24z=0



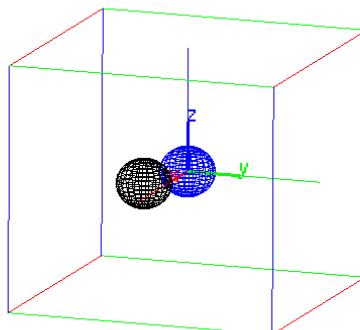
click k: kill 3d view

- *Input:*

```
reflection(line([1,1,0],[-1,-3,0]),point(-1,2,4))
```

Output:

mouse plan 0.916x+0.322y+0.24z=0



click k: kill 3d view

14.14.4 Rotation in space: rotation

See Section 13.15.4 p.953 for rotations in the plane.

The `rotation` command creates a rotation.

- `rotation` takes two mandatory arguments and one optional argument:
 - L , a line (to rotate about).
 - θ , the angle of rotation.
 - Optionally, G , a geometric object.
- `rotation(L, θ)` returns a new command which rotates about L through an angle of θ .
- `reflection(L, θ, G)` returns and draws the rotation of G about L through an angle of θ .

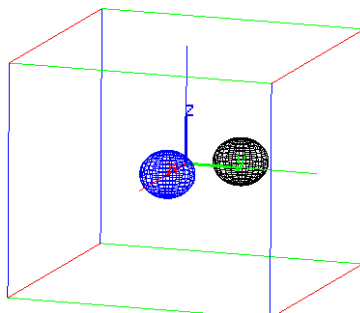
Examples.

- *Input:*

```
S:= sphere([1,0,0],0.5) r:=
rotation(line(point(0,0,0),point(0,0,1)), 2*pi/3)
color(S,blue),r(S)
```

Output:

mouse plan 0.916x+0.322y+0.24z=0



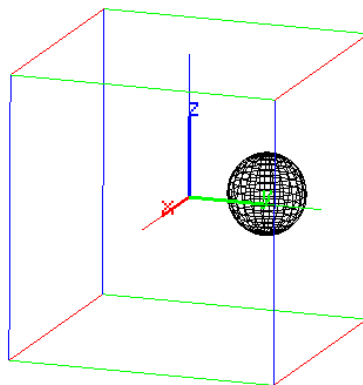
click k: kill 3d view

- *Input:*

```
rotation(line(point(0,0,0),point(0,0,1)), 2*pi/3,S)
```

Output:

```
mouse plan 0.916x+0.322y+0.24z=0
```



```
click k: kill 3d view
```

14.14.5 Homothety in space: homothety

See Section 13.15.5 p.954 for homotheties in the plane.

A homothety is a dilation about a given point. The `homothety` command creates a homothety.

- `homothety` takes two mandatory arguments and one optional argument:
 - P , a point (the center of the homothety).
 - r , a number (the scaling ratio).
 - Optionally, G , a geometric object.
- `homothety(P, r)` returns a new command which dilates about P by a factor of r . If r is complex, this will rotate as well as scale.
- `homothety(P, r, G)` returns and draws the dilation of G about P by a factor of r .

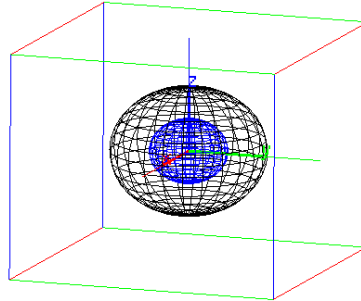
Examples.

- *Input:*

```
h:= homothety(point(0,0,0), 2) S:=sphere([0,0,0],0.5)
color(S,blue),h(S)
```

Output:

mouse plan 0.916x+0.322y+0.24z=0



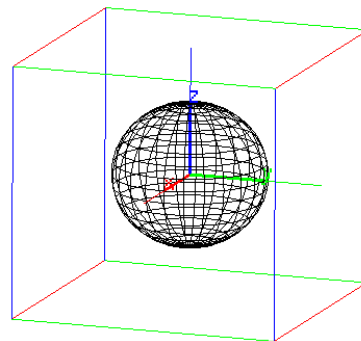
click k: kill 3d view

• *Input:*

homothety(point(0,0,0), 2, S)

Output:

mouse plan 0.916x+0.322y+0.24z=0



click k: kill 3d view

14.14.6 Similarity in space: similarity

See Section 13.15.6 p.955 for similarities in the plane.

The **similarity** command creates a command to rotate and scale about a given line.

- **similarity** takes three mandatory arguments and one optional argument:
 - L , a line (the axis of the rotation).
 - r , a real number (the scaling ratio).
 - θ , a real number (the angle of rotation).
 - Optionally, G , a geometric object.
- **similarity**(L, r, θ) returns a new command which rotates about L through an angle of θ and scales about L by a factor of r . If r is negative, the direction of rotation is reversed.

- `similarity(L,r, θ ,G)` returns and draws the transformation of G .

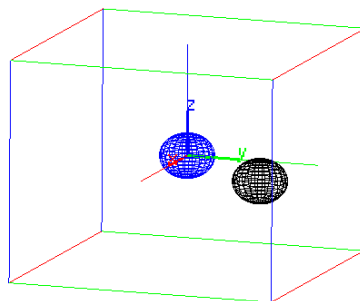
Examples.

- *Input:*

```
S:=sphere([0,0,0],0.5)
s:= similarity(line(point(0,1,0),point(0,1,1)), 2, 2*pi/3)
color(S,blue),s(S)
```

Output:

mouse plan 0.916x+0.322y+0.24z=0



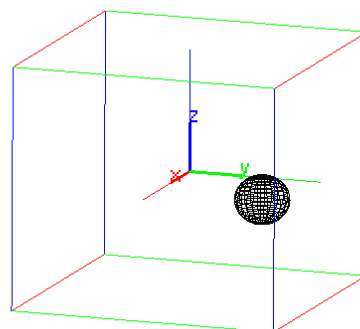
click k: kill 3d view

- *Input:*

```
similarity(line(point(0,1,0),point(1,1,1)), 2, 2*pi/3, S)
```

Output:

mouse plan 0.916x+0.322y+0.24z=0



click k: kill 3d view

14.14.7 Inversion in space: inversion

See Section 13.15.7 p.957 for inversions in the plane.

Given a point P and a real number k , the corresponding *inversion* of a point A is the point A' on the ray \overrightarrow{PA} satisfying $\overline{PA} \cdot \overline{PA'} = k^2$. The `inversion` command creates inversions.

- `inversion` takes two mandatory and one optional argument:
 - P , a point.
 - k , the inversion ratio.
 - Optionally, G , a geometric object.
- `inversion(P,k)` returns a new command which does an inversion about P with a ratio k .
- `inversion(P,k,G)` returns and draws the inversion of G .

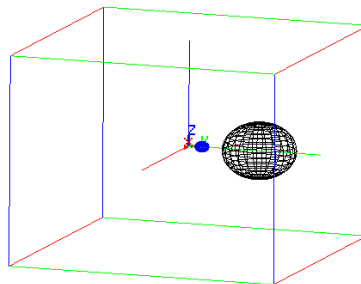
Examples.

- *Input:*

```
S := sphere([0,1,0],0.5)
inver:= inversion(point(0,0,0), 2)
color(S,blue),inver(S)
```

Output:

mouse plan 0.916x+0.322y+0.24z=0

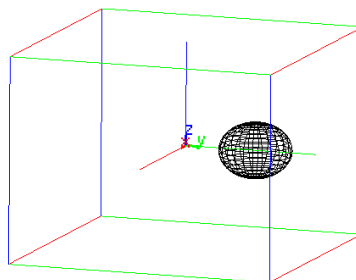


- *Input:*

```
inversion(point(0,0,0),2,S)
```

Output:

mouse plan 0.916x+0.322y+0.24z=0



14.14.8 Orthogonal projection in space: `projection`

See Section 13.15.8 p.959 for projections in the plane.

The `projection` command creates a projection.

- `projection` takes one mandatory argument and one optional argument:
 - O , a geometrix object.
 - Optionally, P , a point.
- `projection(O)` returns a new command which projects points onto O .
- `projection(O, P)` returns and draws the projection of P onto O .

Examples.

- *Input:*

```
P:=point(0,0,1); p1:= projection(line(point(0,0,0),
                                     point(1,1,1))) coordinates(p1(P))
```

Output:

$$\left[\frac{1}{3}, \frac{1}{3}, \frac{1}{3} \right]$$

which is the projection of $(0,0,1)$ onto the line.

- *Input:*

```
coordinates(projection(plane(point(1,0,0),point(0,0,0),point(1,1,1)),point(0,0,1)))
```

Output:

$$\left[0, \frac{1}{2}, \frac{1}{2} \right]$$

which is the projection of the point $(0,0,1)$ onto the plane.

14.15 Surfaces

14.15.1 Cones: `cone`

The `cone` command creates cones.

- `cone` takes three arguments:
 - A , a point.
 - v , a direction vector.
 - θ , a real number.

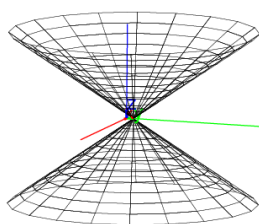
- `cone(A, v, θ)` returns and draws the cone with vertex A , opening in the direction v with an aperture of 2θ .

Example.

Input:

```
cone([0,1,0],[0,0,1],pi/3)
```

Output:



14.15.2 Half-cones: `half_cone`

The `half_cone` command creates half cones.

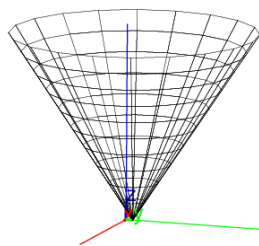
- `half_cone` takes three arguments:
 - A , a point.
 - v , a direction vector.
 - θ , a real number.
- `half_cone(A, v, θ)` returns and draws the half cone with vertex A , opening in the direction v with an aperture of 2θ .

Example.

Input:

```
half_cone([0,1,0],[0,0,1],pi/3)
```

Output:



14.15.3 Cylinders: `cylinder`

The `cylinder` command creates cylinders.

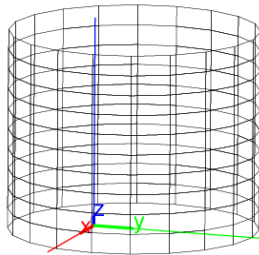
- `cylinder` takes three arguments:
 - A , a point.
 - v , a direction vector.
 - r , a real number.
- `cylinder(A, v, r)` returns and draws the cylinder with axis through A in the direction v with a radius of r . 2θ .

Example.

Input:

```
cylinder([0,1,0],[0,0,1],3)
```

Output:



14.15.4 Spheres: `sphere`

The `sphere` command creates spheres.

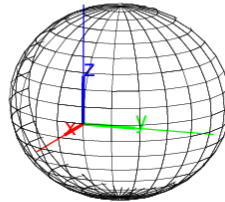
- `sphere` takes two arguments:
 - P, R , either two points or a point and a real number.
- `sphere(P, R)` returns:
 - a sphere with diameter PR , if R is a point.
 - a sphere with center P and radius R , if R is a number.

Examples.

- *Input:*

```
sphere([-2,0,0],[2,0,0])
```

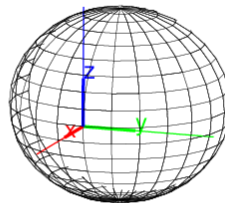
Output:



- *Input:*

```
sphere([0,0,0],2)
```

Output:



14.15.5 The graph of a function of two variables: `funcplot`

The `funcplot` can draw the graphs of two variable functions (see Section 8.4.2 p.668 for a full description).

`plotfunc` is a synonym for `funcplot`.

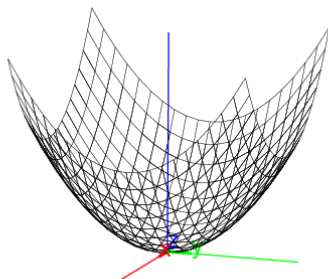
`funcplot` can take two arguments, an expression with two variables and a list of the two variables (possibly with bounds) and it returns and draws the graph of the expression.

Example.

Input:

```
funcplot(x^2 + y^2, [x,y])
```

Output:



14.15.6 The graph of parametric equations in space: `paramplot`

The `paramplot` command can draw a parametric surface in \mathbb{R}^3 (see Section 8.14.2 p.693 for a full description).

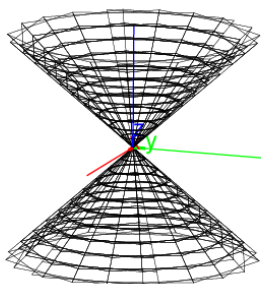
`paramplot` can take two arguments; a list of three expressions involving two parameters and a list of the parameters (possibly with bounds), and it returns and draws the parameterized surface.

Example.

Input:

```
paramplot([u*cos(v), u*sin(v), u], [u, v])
```

Output:



14.16 Solids

14.16.1 Cubes: `cube`

The `cube` command creates cubes.

- `cube` takes three arguments:
 A, B, C , three points.

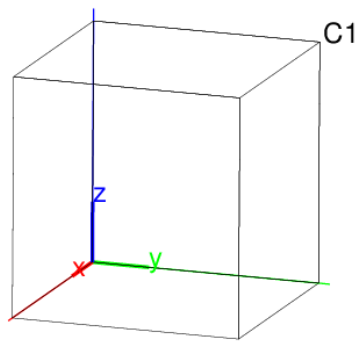
- `cube(A,B,C)` returns and draws the following cube:
 - One edge is AB .
 - One face is in the plane ABC , on the same side of line AB as is C .
 - The cube is on the side of plane ABC that makes the points A , B and C counterclockwise.

Examples.

- *Input:*

```
C1:= cube([0,0,0],[0,4,0],[0,0,1])
```

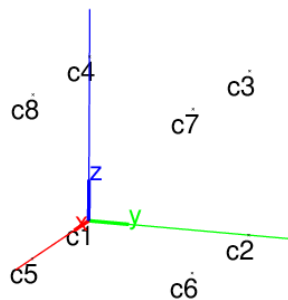
Output:



Input:

```
c1,c2,c3,c4,c5,c6,c7,c8:= vertices(C1)
```

Output:



Input:

```
faces(C1)
```

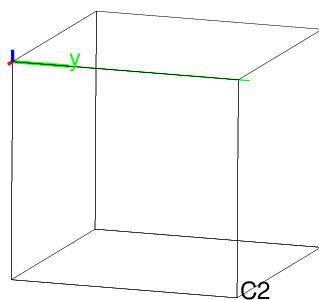
Output:

```
[[[0, 0, 0], [0, 4, 0], [0, 4, 4], [0, 0, 4]], [[4, 0, 0], [4, 4, 0], [4, 4, 4], [4, 0, 4]], [[0, 0, 0], [4, 0, 0], [4, 0, 4], [0, 0, 4]]]
```

• *Input:*

```
C2:= cube([0,0,0],[0,4,0],[0,0,-1])
```

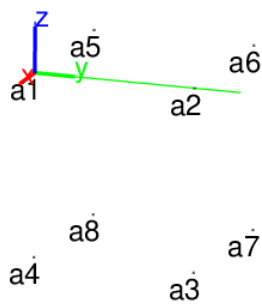
Output:



Input:

```
a1,a2,a3,a4,a5,a6,a7,a8:= vertices(C2)
```

Output:



14.16.2 Tetrahedrons: tetrahedron pyramid

The `tetrahedron` creates tetrahedra.

`pyramid` is a synonym for `tetrahedron`.

- `tetrahedron` command takes three or four arguments:
 - A, B, C , three points.
 - Optionally D , another point.
- `tetrahedron(A, B, C)` returns and draws the regular tetrahedron given by:
 - One edge is AB .
 - One face is in the plane ABC , on the same side of line AB as is C .
 - The tetrahedron is on the side of plane ABC that makes the points A, B and C counterclockwise.
- `tetrahedron(A, B, C, D)` returns and draws the tetrahedron $ABCD$.

Examples.

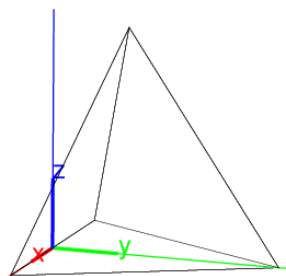
- *Input:*

```
tetrahedron([-2,0,0],[2,0,0],[0,2,0])
```

or:

```
pyramid([-2,0,0],[2,0,0],[0,2,0])
```

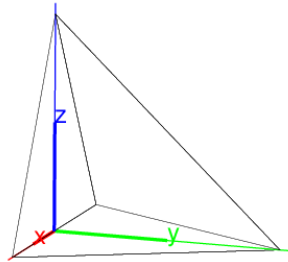
Output:



- *Input:*

```
tetrahedron([-2,0,0],[2,0,0],[0,2,0],[0,0,2])
```

Output:



14.16.3 Parallelepipeds: `parallelepiped`

The `parallelepiped` command creates parallelepipeds.

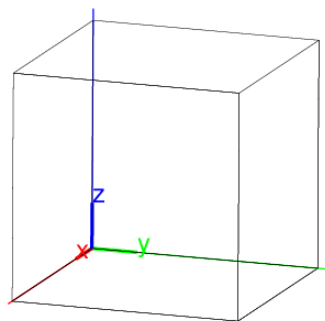
- `parallelepiped` takes four arguments:
 A, B, C, D , four points.
- `parallelepiped(A, B, C, D)` returns and draws the parallelepiped determined by the edges AB , AC and AD .

Examples.

- *Input:*

```
parallelepiped([0,0,0],[5,0,0],[0,5,0],[0,0,5])
```

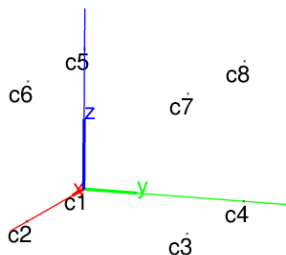
Output:



- *Input:*

```
p:= parallelepiped([0,0,0],[5,0,0],[0,3,0],[0,0,2]);
c1, c2, c3, c4, c5, c6, c7, c8:= vertices(p);
```

Output:



14.16.4 Prisms: prism

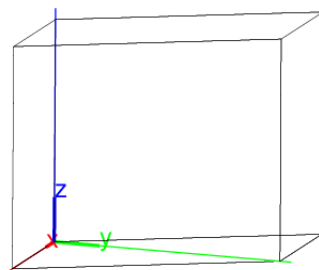
The **prism** command takes two arguments, a list of coplanar points $[A, B, \dots]$ and an additional point $A1$.

prism returns and draws the prism whose base is the polygon determined by the points A, B, \dots , and with edges parallel to $AA1$.

Input:

```
prism([[0,0,0],[5,0,0],[0,5,0],[-5,5,0]],[0,0,5])
```

Output:



14.16.5 Polyhedra: polyhedron

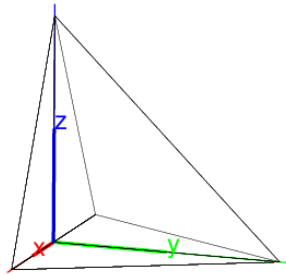
The **polyhedron** command takes as argument a sequence of points.

polyhedron returns and draws the convex polyhedron whose vertices are from the list of points such that the remaining points are inside or on the surface of the polyhedron.

Input:

```
polyhedron([0,0,0],[-2,0,0],[2,0,0],[0,2,0],[0,0,2])
```

Output:



14.16.6 Vertices: vertices

The **vertices** command takes as argument a polyhedron.

vertices returns and draws a list of the vertices of the polyhedron.

Input:

```
V:=
vertices(polyhedron([0,0,0],[-2,0,0],[2,0,0],[0,2,0],[0,0,2]))
```

then:

```
coordinates(V)
```

Output:

```
[[0,0,0],[-2,0,0],[2,0,0],[0,2,0],[0,0,2]]
```

14.16.7 Faces: faces

The **faces** command takes as argument a polyhedron.

faces returns a list of the faces of the polyhedron.

Input:

```
faces(polyhedron([1,-1,0],[1,1,0],[0,0,2],[0,0,-2],[-1,1,0],[-1,-1,0]))
```

Output:

```
[[[1,-1,0],[1,1,0],[0,0,2]],[[1,-1,0],[1,1,0],[0,0,-2]],
 [[1,-1,0],[0,0,2],[-1,-1,0]],[[1,-1,0],[0,0,-2],[-1,-1,0]],
 [[1,1,0],[0,0,2],[-1,1,0]],[[1,1,0],[0,0,-2],[-1,1,0]],
 [[0,0,2],[-1,1,0],[-1,-1,0]],[[0,0,-2],[-1,1,0],[-1,-1,0]]]
```

14.16.8 Edges: `line_segments`

The `line_segments` command takes as argument a polyhedron.

`line_segments` returns and draws a list of the edges of the polyhedron.

Input:

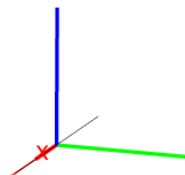
```
line_segments(polyhedron([0,0,0],[-2,0,0],[2,0,0],[0,2,0],[0,0,2]))
```

Output:

Input:

```
line_segments(polyhedron([0,0,0],[-2,0,0],[2,0,0],[0,2,0],[0,0,2]))[1]
```

Output:



14.17 Platonic solids

To specify a Platonic solid, `Xcas` works with the center, a vertex and a third point to specify a plane of symmetry. To speed up calculations, it may be useful to use approximate calculations, which can be ensured with the `evalf` command. For example, instead of:

Input:

```
centered_cube([0,0,0],[3,2,1],[1,1,0])
```

it would typically be better to use:

Input:

```
centered_cube(evalf([0,0,0],[3,2,1],[1,1,0]))
```

14.17.1 Centered tetrahedra: centered_tetrahedron

The `centered_tetrahedron` command creates a regular tetrahedron.

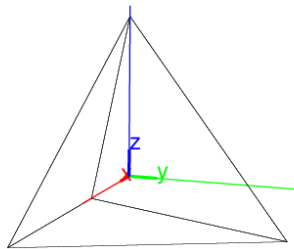
- `centered_tetrahedron` command takes three arguments:
 A, B, C , three points.
- `centered_tetrahedron(A, B, C)` returns and draws the tetrahedron centered at A , with a vertex at B and another vertex on the plane ABC .

Example.

Input:

```
centered_tetrahedron([0,0,0],[0,0,6],[0,1,0])
```

Output:

**14.17.2 Centered cubes: centered_cube**

The `centered_cube` command draws a cube.

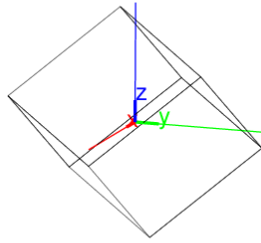
- `centered_cube` takes three arguments:
 A, B, C , three points.
- `centered_cube(A, B, C)` returns and draws the cube centered at A with a vertex at B and plane of symmetry ABC . This plane of symmetry has an edge of the cube containing B , the other endpoint of this edge is on the same side of line AB as C .

Examples.

- *Input:*

```
centered_cube([0,0,0],[3,3,3],[0,1,0])
```

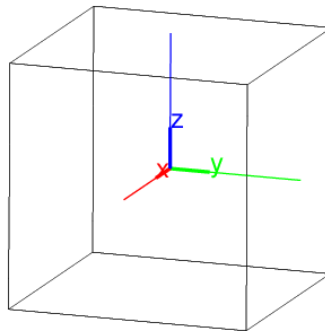
Output:



- *Input:*

```
centered_cube([0,0,0],[3,3,3],[0,-1,0])
```

Output:



Note that there are two cubes centered at A with a vertex at B and with a plane of symmetry ABC . Each cube has an edge containing B that's contained in plane of symmetry, these edges are on opposite sides of the line AB . The cube that `cube(A,B,C)` returns is the cube whose edge is on the same side of AB as the point C .

14.17.3 Octahedra: `octahedron`

The `octahedron` command creates regular octahedra.

- `octahedron` takes three arguments:
 A, B, C , three points.
- `octahedron(A,B,C)` returns and draws the octahedron centered at A which has a vertex at B and with four vertices in the plane ABC .

Example.*Input:*

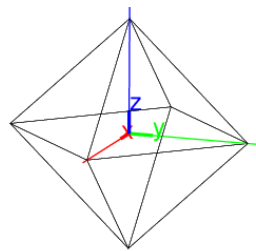
```
octahedron([0,0,0],[0,0,5],[0,1,0])
```

or:

```
octahedron([0,0,0],[0,5,0],[0,0,1])
```

or:

```
octahedron([0,0,0],[5,0,0],[0,0,1])
```

Output:**14.17.4 Dodecahedra: dodecahedron**

The `dodecahedron` command creates a regular dodecahedron.

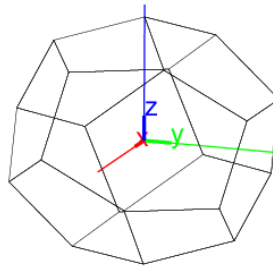
- `dodecahedron` takes three arguments:
 A, B, C , three points.
- `dodecahedron(A, B, C)` returns and draws the dodecahedron centered at A with a vertex at B and with an axis of symmetry in the plane ABC . (Note that each face is a pentagon, but will be drawn with one of its diagonals and so will show up as a trapezoid and a triangle.)

Examples.

- *Input:*

```
dodecahedron([0,0,0],[0,0,5],[0,1,0])
```

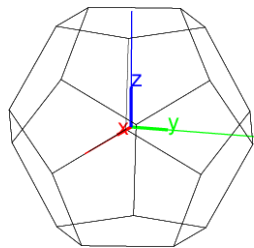
Output:



- *Input:*

```
dodecahedron([0,0,0],[0,2,sqrt(5)/2 + 3/2],[0,0,1])
```

Output:



14.17.5 Icosahedra: icosahedron

The `icosahedron` command creates regular icosahedra.

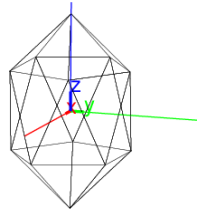
- `icosahedron` takes three arguments:
 A, B, C , three points.
- `icosahedron(A, B, C)` returns and draws the icosahedron centered at A with a vertex at B and such that the plane ABC contains one of the vertices closest to B .

Examples.

- *Input:*

```
icosahedron([0,0,0],[0,0,5],[0,1,0])
```

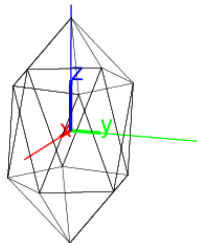
Output:



• *Input:*

```
icosahedron([0,0,0],[0,0,sqrt(5)],[2,1,0])
```

Output:



Chapter 15

Multimedia

15.1 Audio Tools

Xcas has commands for working with audio objects. An audio object is a vector consisting of:

- The first element is itself a list consisting of:
 - The number of channels (generally 1 for mono and 2 for stereo).
 - The number of bits (generally 16).
 - The sampling frequency (44100 for a CD quality sound).
 - The number of bytes (excluding the header); i.e., the number of seconds times the sampling frequency times the number of bits/8 times the number of channels.
- The second element is another list of digital sound data for each channel.

Xcas can read and write audio objects as files on your computer; these files will be in the **wav** (Waveform Audio File) format.

For creating and playing audio objects, there are:

- **creawav**, for creating audio objects (see Section 15.1.1 p.1070).
- **playsnd**, for playing audio objects (see Section 15.1.4 p.1072).

For reading and writing audio files, there are:

- **readwav**, for reading audio files from disk and creating audio objects (see Section 15.1.2 p.1071).
- **writewav**, for writing audio files to disk from audio objects (see Section 15.1.3 p.1071).

For manipulating audio objects, there are:

- **stereo2mono**, to convert a multichannel audio clip to a single channel (see Section 15.1.5 p.1072).
- **resample**, to change the sample rate (see Section 15.1.8 p.1074).

For getting information from an audio object, there are:

- `channels`, to find the number of channels (see Section 15.1.6 p.1073).
- `bit_depth`, to find the number of bits in each sample value (see Section 15.1.6 p.1073).
- `samplerate`, to find the number of samples per second (see Section 15.1.6 p.1073).
- `duration`, to find the duration of the audio in seconds (see Section 15.1.6 p.1073).
- `channel_data`, to extract a sample from an audio (see Section 15.1.7 p.1073).
- `plotwav`, to visualize a waveform (see Section 15.1.9 p.1075).
- `plotspectrum`, to visualize the power spectra (see Section 15.1.10 p.1076).

15.1.1 Creating audio clips: `createwav`

The `createwav` command creates an audio object with specified parameters.

- `createwav` takes its arguments as `key=value` pairs, in no particular order. The following arguments are all optional:
 - `size=n` resp. `duration=T`, where n resp. T is the total number of samples resp. the length in seconds.
 - `bit_depth=b`, where b is the number of bits reserved for each sample value and may be 8 or 16 (by default 16).
 - `samplerate=r`, where r is the number of samples per second (by default 44100).
 - `channels=c` where c is the number of channels (by default 1).
 - `D` or `channel_data=D`, where D is a list or a matrix.
 If D is a matrix, it should contain the k -th sample in the j -th channel at position (j, k) . The value of each sample must be a real number in range $[-1.0, 1.0]$. Any value outside this interval is clamped to it (the resulting effect is called *clipping*).
 If D is a single list, it is copied across channels.
 D may be truncated or padded with zeros to match the appropriate number of samples or seconds.
 - `normalize=db`, where $db \leq 0$ is a real number representing the amplitude peak level in dB FS (decibel "full scale") units. If this option is given, audio data is normalized to the specified level prior to conversion. This can be used to avoid clipping.
- `createwav(<keys=values>)` returns an audio object with the requested parameters.

Examples.

(See Section 15.1.4 p.1072 for a description of `playsnd` and Section 15.1.14 p.1078 for a description of `soundsec`):

- *Input:*

```
wave:=sin(2*pi*440*soundsec(2))
s:=creawav(channel_data=wave,samplerate=48000)
playsnd(s)
```

Output:

Two seconds of the 440 Hz sine wave at rate 48000.

- *Input:*

```
t:=soundsec(3)
L,R:=sin(2*pi*440*t),sin(2*pi*445*t)
s:=creawav([L,R])
playsnd(s)
```

Output:

Three seconds of a vibrato effect on a sine wave (stereo).

15.1.2 Reading wav files from disk: readwav

The `readwav` command loads a `wav` file.

- `readwav` takes one argument:
file, a string containing the name of a `wav` file.
- `readwav(file)` loads *file* and returns an audio clip object.

Example.

(Assuming that the file `example.wav` is stored in the directory `sounds`)

Input:

```
s:=readwav("/path/to/sounds/example.wav")
```

You can now play the audio clip object `s`:

Input:

```
playsnd(s)
```

Output:

The sound of the audio file `example.wav`.

15.1.3 Writing wav files to disk: writewav

The `writewav` command writes `wav` files to disk.

- `writewav` takes two arguments:
 - *filename*, a string containing a file name.

- A , an audio clip object.
- `writewav(filename, A)` writes the clip A as a `wav` file named *filename*. It returns 1 on success and 0 on failure.

Example.*Input:*

```
s:=createwav(sin(2*pi*440*soundsec(1)));;
writewav("sounds/sine.wav",s)
```

Output:

1

The `sounds` directory will contain the file `sine.wav`.

15.1.4 Audio playback: playsnd

The `playsnd` command plays audio clips.

- `playsnd` takes one argument:
 A , an audio clip.
- `playsnd(A)` plays the audio clip A .

Example.*Input:*

```
playsnd(createwav(sin(2*pi*440*soundsec(3))))
```

Output: The sound of the sine wave will be played.**15.1.5 Averaging channel data: stereo2mono**

The `stereo2mono` command converts a multichannel audio clip to a single channel audio clip.

- `stereo2mono` takes one argument:
 A , a multichannel audio clip.
- `stereo2mono(A)` returns an audio clip with the input channels in A mixed down to a single channel.
Every sample in the output is the arithmetic mean of the samples at the same position in the input channels.

Example.*Input:*

```
t:=soundsec(3);;
L,R:=sin(2*pi*440*t),sin(2*pi*445*t);;
s:=stereo2mono(createwav([L,R]));;
playsnd(s)
```

Output: The sound of the single channel of the combination of L and R is played.

15.1.6 Audio clip properties: `channels` `bit_depth` `samplerate` `duration`

The `channels`, `bit_depth`, `samplerate` and `duration` commands find properties of an audio clip.

- `channels`, `bit_depth`, `samplerate` and `duration` each take one argument:
 A , an audio clip.
- `channels(A)` returns the number of channels in A .
- `bit_depth(A)` returns the number of bits reserved for each sample value (8 or 16) in A .
- `samplerate(A)` returns the number of samples per second in A .
- `duration(A)` returns the duration of A in seconds.

15.1.7 Extracting samples from audio clips: `channel_data`

The `channel_data` command gets samples from an audio clip.

- `channel_data` takes one mandatory argument and up to two optional argument:
 - A , an audio clip.
 - Optionally `options`, which can be the following (the order is unimportant):
 - * One of:
 - n , a positive integer (channel number).
 - `matrix`, the symbol.
 - * One of:
 - `range= $[m, n]$` or `range= $m..n$` for nonnegative integers m and n .
 - `range= $a..b$` for floating point numbers a and b .
- `channel_data($A \langle, options \rangle$)` returns data from the channels as a sequence of lists. The returned sample values are all within the interval $[-1.0, 1.0]$, i.e. the amplitude of the returned signal is relative. The maximum possible amplitude is represented by the value 1.0.
 - With no options, data from all channels are returned as a sequence of lists.
 - With the option n (the channel number) or if there is only one channel, only the data from this channel is returned in a single list.
 - With the option `matrix`, the lists representing the channel data are returned as the rows of a matrix.

- With the option `range=[m,n]` or `range=m..n`, with m and n integers, only the samples from n -th to m -th (inclusive) are extracted.
- With the option `range=a..b`, with floating point numbers a and b , then a and b are bounds in seconds.

Example.

Assuming that the directory `sounds` contains `example.wav`, a `wav` file with three seconds of stereo sound:

Input:

```
s:=readwav("/path/to/sounds/example.wav");;
L,R:=channel_data(s,range=1.2..1.5)
```

Output:

A list `L` resp. `R` containing the data between 1.2 and 1.5 seconds in the left resp. right channel of the original file.

15.1.8 Changing the sampling rate: resample

The `resample` command resamples a clip to a desired rate.

- `resample` takes one mandatory argument and two optional arguments:
 - A , an audio clip.
 - Optionally, r , the target sample rate in Hz (by default 44100).
 - Optionally, n , an integer specifying the quality level, from 0 (poor) to 4 (best) (by default, 2).
- `resample($A \langle r, n \rangle$)` returns the audio clip A resampled to rate r .

`giac` does resampling by using the `libsamplerate` library <http://www.mega-nerd.com/libsamplerate/> written by Erik de Castro Lopo. For more information see the library documentation.

Example.

Assuming that the directory `sounds` contains `example.wav`, a `wav` file with a sample rate of 44100:

Input:

```
clip:=readwav("/path/to/sounds/example.wav");; samplerate(clip)
```

Output:

44100

Input:

```
res:=resample(clip,48000);; samplerate(res)
```

Output:

48000

15.1.9 Visualizing waveforms: `plotwav`

The `plotwav` command displays the waveform of an audio clip.

- `plotwav` takes one mandatory argument and one optional argument:
 - A , an audio clip.
 - Optionally, $range$; one of:
 - * `range=[m, n]` for integers m and n (representing sample units).
 - * `range= $a..b$` for floating point numbers a and b (representing seconds).
- `plotwav(A $\langle range \rangle$)` displays the waveform A , in its entirety or over the optional specified range.

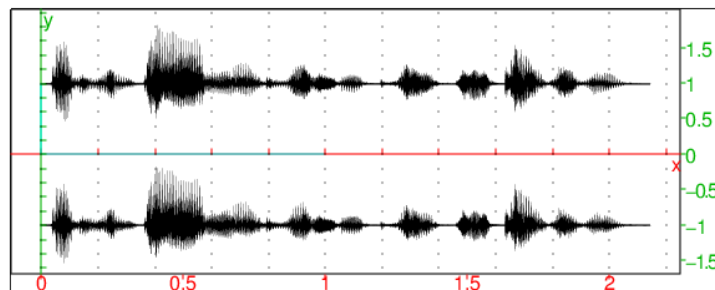
Examples.

Assuming that the directory `sounds` contains two `wav` files, `example1.wav` (a man speaking, stereo) and `example2.wav` (guitar playing, mono):

- *Input:*

```
clip1:=readwav("/path/to/sounds/example1.wav");
plotwav(clip1)
```

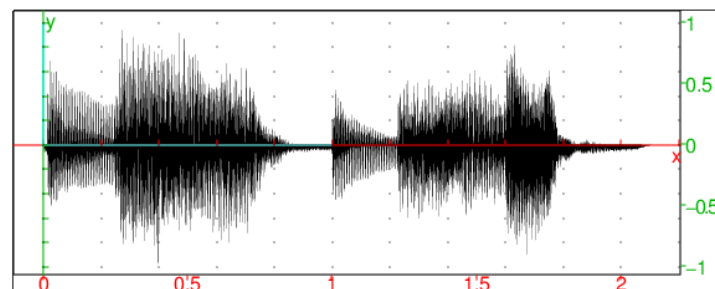
Output:



- *Input:*

```
clip2:=readwav("/path/to/sounds/example2.wav");
plotwav(clip2)
```

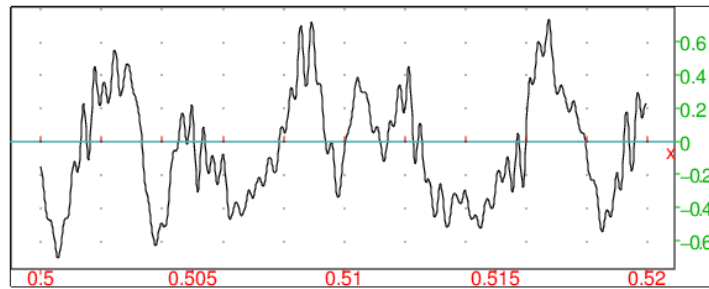
Output:



- *Input:*

```
plotwav(clip2,range=0.5..0.52)
```

Output:



15.1.10 Visualizing power spectra: `plotspectrum`

The `plotspectrum` command displays the power spectrum of an audio clip.

- `plotspectrum` takes one mandatory and one optional argument:
 - A , an audio clip.
 - Optionally, *range*, which can be in the form `range=[lf,uf]` or `range=lf..uf`, where lf is the lower bound and uf the upper bound of the desired frequency band (by default, `range=[0, s/2]` where s is the sampling rate).
- `plotspectrum(A⟨range⟩)` displays the power spectrum of the audio data on the specified frequency range.
If the audio clip has more than one channel, the channels are mixed down to a single channel before computing the spectrum.

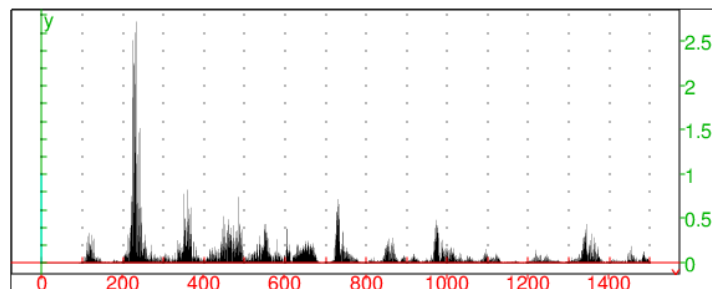
Example.

Assuming that a male voice is recorded in the file `example1.wav`:

Input:

```
clip:=readwav("/path/to/sounds/example1.wav");  
plotspectrum(clip,range=[0,1500])
```

Output:



You can see that the dominant frequency is around 220 Hz, which is the middle of tenor range. This is consistent with the fact that a man is speaking in the clip.

15.1.11 Reading a wav file: readwav

The `readwav` command retrieves information about a `wav` file.

- `readwav` takes one argument:
 - filename*, a sound file (with extension `.wav`) stored in `wav` format given as a string.
- `readwav(filename)` returns a vector consisting of:
 - A list consisting of:
 - * The number of channels (generally 1 for mono and 2 for stereo).
 - * The number of bits (generally 16).
 - * The sampling frequency (44100 for a CD quality sound).
 - * The number of bytes (excluding the header); i.e., the number of seconds times the sampling frequency times the number of bits/8 times the number of channels.
 - A list of digital sound data for each channel.

The result of `readwav` is typically stored in a variable.

Example.

Assuming that `sound.wav` is a sound file for a one-second sound in CD quality on a 16-bit channel:

Input:

```
s:= readwav("sound.wav");
s[0]
```

Output:

```
[1,16,44100,88200]
```

Input:

```
size(s)
```

Output:

```
2
```

which is the number of channels plus 1.

Input:

```
size(s[1])
```

Output:

```
44100
```

15.1.12 Writing a wav file: `writewav`

The `writewav` command writes sound data to a `wav` file.

- `writewav` takes two arguments:
 - *filename*, the name of a file.
 - *s*, the sound data. This can either be in the same format as that returned by the `readwav` command or (for a mono sound) a list of the digital data of the sound which will use the default parameters (16 bits, 44100 Hz).
- `writewav(filename,s)` writes the sound to the file *filename*.

Example.

Input:

```
writewav("la.wav",2^14*sin(2*pi*440*soundsec(1)))
```

Output: There will be a file `la.wav` containing a sound of frequency 440 Hz sampled 44100 times per second.

15.1.13 Listening to a digital sound: `playsnd`

The `playsnd` command plays digitized sound data.

- `playsnd` takes one argument:
 - s*, digitized sound data, such as that which can be read with the `readwav` command or generated with the help of `soundsec`. *s* should be either in the format of the output of the `readwav` command or a list of sampled data for mono sound with the default settings of 1 channel, 16 bits and 44100 Hz.
- `playsnd(s)` plays the given sound.

15.1.14 Preparing digital sound data: `soundsec`

The `soundsec` command prepares sound data in the form of a vector.

- `soundsec` takes one mandatory argument and one optional argument:
 - *d*, a real number (the duration).
 - Optionally, *f*, a real number (the sampling frequency, by default 44100).
- `soundsec(d⟨f⟩)` returns sound data with duration *d* seconds, and with sampling frequency *f*. The sound data is returned as a vector, whose *i*th element is the time corresponding to index *i*.

Examples.

- *Input:*

```
soundsec(2.5)
```

Output:

Sound data 2.5 seconds long sampled at the default frequency of 44100 Hz.

- *Input:*

```
soundsec(1,22050)
```

Output: Sound data 1 second long sampled at the frequency of 22050 Hz.

- *Input:*

```
sin(2*pi*440*soundsec(1.3))
```

Output: A sinusoid with frequency 440 Hz sampled 44100 times per second for 1.3 seconds.

15.2 Signal Processing

15.2.1 Boxcar function: `boxcar`

The `boxcar` command creates a function which is 0 everywhere except in a given interval, where it is 1.

- `boxcar` takes three arguments:
 - a, b , two real numbers.
 - x , an identifier or expression.
- `boxcar(a, b, x)` returns $\theta(x - a) - \theta(x - b)$, where θ is the Heaviside function (see Section 6.8.9 p.179). The resulting expression defines a function which is zero everywhere except within the segment $[a, b]$, where its value is equal to 1.

Examples.

- *Input:*

```
boxcar(1,2,x)
```

Output:

$$\theta(x - 1) - \theta(x - 2)$$

- *Input:*

```
boxcar(1,2,3/2)
```

Output:

- *Input:*

`boxcar(1,2,0)`

Output:

0

15.2.2 Rectangle function: `rect`

The rectangle function Π is 0 everywhere except on $[-1/2, 1/2]$, where it is 1; namely, $\Pi(x) = \theta(x + 1/2) - \theta(x - 1/2)$ where θ is the Heaviside function. The rectangle function is a special case of boxcar function (see section 15.2.1) for $a = -\frac{1}{2}$ and $b = \frac{1}{2}$.

- `rect` takes one argument:
 x , an identifier or an expression.
- `rect(x)` returns the value of the rectangle function at x .

Example.

Input:

`rect(x/2)`

Output:

$$\theta\left(\frac{x}{2} + \frac{1}{2}\right) - \theta\left(\frac{x}{2} - \frac{1}{2}\right)$$

To compute the convolution of the rectangle function with itself, you can use the convolution theorem.

Input:

`R:=fourier(rect(x),x,s); ifourier(R^2,s,x)`

Output:

$$-2x\theta(x) + x\theta(x+1) + x\theta(x-1) + \theta(x+1) - \theta(x-1)$$

This result is the triangle function $\text{tri}(x)$ (see section 15.2.3).

15.2.3 Triangle function: `tri`

The triangle function is defined by

$$\Lambda(x) = \begin{cases} 1 - |x|, & |x| < 1, \\ 0, & \text{otherwise.} \end{cases}$$

This is equal to the convolution of rectangle function with itself (see Section 15.2.2).

The `tri` command computes the triangle function.

- `tri` takes one argument:
 x , an expression.

- `tri(x)` returns the value of triangle function at x .

Example.

Input:

`tri(x-1)`

Output:

$$(-x+1+1)(\theta(x-1)-\theta(x-1-1))+(1+x-1)(\theta(-x+1)-\theta(-x+1-1))$$

15.2.4 Cardinal sine function: `sinc`

The `sinc` function is defined by

$$\text{sinc}(x) = \begin{cases} \frac{\sin(x)}{x}, & x \neq 0, \\ 1, & x = 0. \end{cases}$$

The `sinc` command computes the sinc function.

- `sinc` takes one argument:
 x , an expression.
- `sinc(x)` returns the value of the sinc function at x .

Examples.

- *Input:*

`sinc(pi*x)`

Output:

$$\frac{\sin(\pi x)}{\pi x}$$

- *Input:*

`sinc(0)`

Output:

1

15.2.5 Root mean square: `rms`

The `rms` command finds the root mean square of a list of numbers $X = [x_1, x_2, \dots, x_n]$, which is defined by

$$\text{RMS}(X) = \sqrt{\frac{\sum_{k=1}^n |x_k|^2}{n}}.$$

- `rms` takes one argument:
 X , a list of real or complex numbers $[x_1, x_2, \dots, x_n]$.

- `rms(X)` returns the root mean square of X .

Examples.

- *Input:*

```
rms([1,2,5,8,3,6,7,9,-1])
```

Output:

$$\sqrt{30}$$

- *Input:*

```
rms([1,1-i,2+3i,5-2i])
```

Output:

$$\frac{3}{2}\sqrt{5}$$

15.2.6 Cross-correlation of two signals: `cross_correlation`

The cross correlation of two complex vectors $v = [v_1, \dots, v_n]$ and $w = [w_1, \dots, w_m]$ is the complex vector $z = v \star w$ of length $n + m - 1$ given by

$$z_k = \sum_{i=k}^{N-1} \overline{v_{i-k}^*} w_i^*, \quad k = 0, 1, \dots, N-1,$$

where

$$v^* = [v_0, v_1, \dots, v_{n-1}, \underbrace{0, 0, \dots, 0}_{m-1}] \quad \text{and} \quad w^* = [\underbrace{0, 0, \dots, 0}_{n-1}, w_0, w_1, \dots, w_{m-1}].$$

Cross-correlation is typically used for measuring similarity between signals.

The `cross_correlation` command computes the cross correlation of two vectors.

- `cross_correlation` takes two arguments:
 v, w , two vectors (not necessarily the same length).
- `cross_correlation(v, w)` returns the cross correlation $v \star w$.

Examples.

- *Input:*

```
cross_correlation([1,2],[3,4,5])
```

Output:

$$[6.0, 11.0, 14.0, 5.0]$$

- *Input:*

```
v:=[2,1,3,2]:: w:=[1,-1,1,2,2,1,3,2,1]::
round(cross_correlation(v,w))
```

Output:

```
[2, 1, 0, 8, 9, 12, 15, 18, 13, 11, 5, 2]
```

Observe that the cross-correlation of \mathbf{v} and \mathbf{w} is peaking at position 8 with the value 18, indicating that the two signals are best correlated when the last sample in \mathbf{v} is aligned with the eighth sample in \mathbf{w} . Indeed, there is an occurrence of \mathbf{v} in \mathbf{w} precisely at that point.

15.2.7 Auto-correlation of a signal: `auto_correlation`

The auto correlation of a vector is its cross correlation with itself (see Section 15.2.6 p.1082). The `auto_correlation` command computes the auto correlation of a vector.

- `auto_correlation` takes one argument:
 v , a complex vector.
- `auto_correlation(v)` returns the cross-correlation of v with itself, $v \star v$.

Example.

Input:

```
auto_correlation([2,3,4,3,1,4,5,1,3,1])
```

Output:

```
[2.0, 9.0, 15.0, 28.0, 37.0, 44.0, 58.0, 58.0, 68.0, 91.0, 68.0, 58.0, 58.0, 44.0, 37.0, 28.0, 15.0, 9.0, 2.0]
```

15.2.8 Convolution of two signals or functions: `convolution`

The convolution of two real vectors $v = [v_1, \dots, v_n]$ and $w = [w_1, \dots, w_m]$ is the complex vector $z = v * w$ of length $n + m - 1$ given by

$$z_k = \sum_{i=0}^k v_i w_{k-i}, \quad k = 0, 1, \dots, N-1,$$

such that $v_j = 0$ for $j \geq n$ and $w_j = 0$ for $j \geq m$.

The convolution of two real functions $f(x)$ and $g(x)$ is the integral

$$\int_{-\infty}^{+\infty} f(t) g(x-t) dt$$

variable x as an optional third argument, in which case the `convolution` takes two arguments, a real vector \mathbf{v} of length n and a real vector \mathbf{w} of length m , and returns their convolution $\mathbf{z} = \mathbf{v} * \mathbf{w}$ which is the vector of length $N = n + m - 1$ defined as:

The `convolution` command finds the convolution of two vectors or two functions.

For the convolution of two vectors:

- `convolution` takes two arguments:
 v, w , two vectors (not necessarily the same length).
- `convolution(v, w)` returns the convolution correlation $v * w$.

Example.*Input:*

```
convolution([1,2,3],[1,-1,1,-1])
```

Output:

```
[1.0, 1.0, 2.0, -2.0, 1.0, -3.0]
```

For the convolution of two functions:

- `convolution` takes two mandatory arguments and one optional argument:
 - f, g , two expressions of a variable.
 - Optionally, x , the variable name.
- `convolution(f, g(x))` returns the convolution of f and g .
 The functions f and g are causal functions, i.e. $f(x) = g(x) = 0$ for $x < 0$. Therefore both f and g are multiplied by Heaviside function prior to integration.

Examples.

- Compute the convolution of $f(x) = 25e^{2x}u(x)$ and $g(x) = xe^{-3x}\theta(x)$, where θ is the Heaviside function:

Input:

```
convolution(25*exp(2x),x*exp(-3x))
```

Output:

$$(-5xe^{-3x} - e^{-3x} + e^{2x})\theta(x)$$

- Compute the convolution of $f(t) = \ln(1+t)u(t)$ and $g(t) = \frac{1}{\sqrt{t}}$.

Input:

```
convolution(ln(1+t),1/sqrt(t),t)
```

Output:

$$-\frac{\left(2t \ln\left(\frac{|2\sqrt{t}-2\sqrt{t+1}|}{2\sqrt{t}+2\sqrt{t+1}}\right) + 4\sqrt{t}\sqrt{t+1} + 2 \ln\left(\frac{|2\sqrt{t}-2\sqrt{t+1}|}{2\sqrt{t}+2\sqrt{t+1}}\right)\right)}{\sqrt{t+1}}\theta(t)$$

- In this example, convolution is used for reverberation. Assume that the directory `sounds` contains two files, a dry, mono recording of a guitar stored in `guitar.wav` and a two-channel impulse response recorded in a French 18th century salon and stored in `salon-ir.wav`.

Load the files:

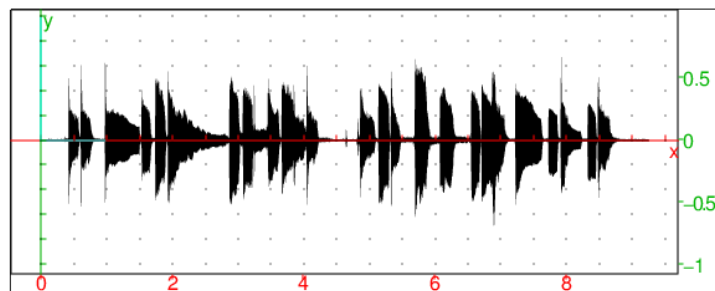
Input:

```
clip:=readwav("/path/to/sounds/guitar.wav");;
ir:=readwav("/path/to/sounds/salon-ir.wav");;
```

Then: *Input:*

```
plotwav(clip)
```

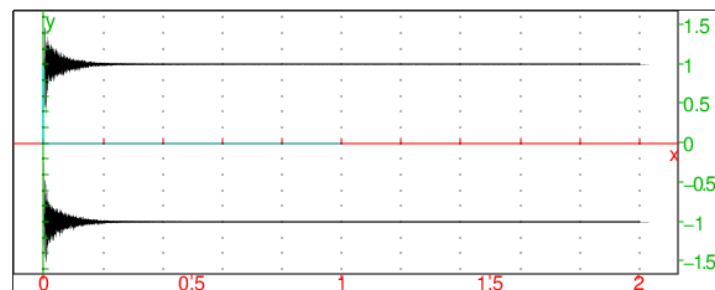
Output:



Input:

```
plotwav(ir)
```

Output:



Convoluting the data from `clip` with both channels in `ir` produces a reverberated variant of the recording, in stereo.

Input:

```
data:=channel_data(clip);;
L:=convolution(data,channel_data(ir,1));;
R:=convolution(data,channel_data(ir,2));;
```

The convolved signals `L` and `R` now become the left and right channel of a new audio clip, respectively. The `normalize` option is used because convolution usually results in a huge increase of sample values (which is clear from the definition).

- *Input:*

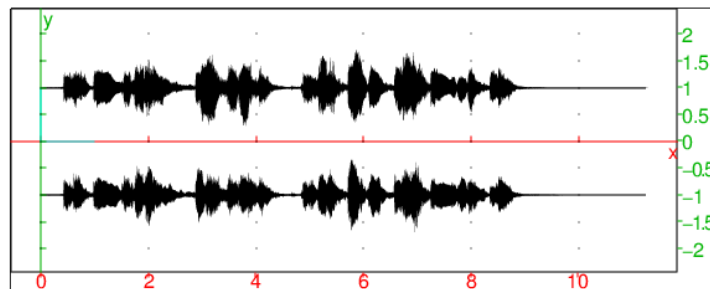
```
spatial:=createwav([L,R],normalize=-3);; playsnd(spatial)
```

Output: A sound that sounds as if it was recorded in the same salon as the impulse response. Furthermore, it is a true stereo sound. To visualize it:

Input:

```
plotwav(spatial)
```

Output:



Note that the resulting audio is longer than the input (for the length of the impulse response).

15.2.9 Low-pass filtering: `lowpass`

The `lowpass` command applies a simple first-order lowpass RC filter to an audio clip.

- `lowpass` takes two mandatory arguments and one optional argument:
 - A , an audio clip or a real vector representing the sampled signal.
 - c , a real number specifying the cutoff frequency.
 - Optionally, r , a samplerate (by default 44100).
- `lowpass($A, c \langle r \rangle$)` returns the input data after applying a simple first-order lowpass RC filter.

Example.

Input:

```
f:=unapply(periodic(sign(x),x,-1/880,1/880),x);
s:=apply(f,soundsec(3));
playsnd(lowpass(createwav(s),1000))
```

Output:

The sound of the periodic signal after a simple first-order lowpass RC filter has been applied.

15.2.10 High-pass filtering: `highpass`

The `highpass` command applies a simple first-order lowpass RC filter to an audio clip.

- `highpass` takes two mandatory arguments and one optional argument:
 - A , an audio clip or a real vector representing the sampled signal.
 - c , a real number specifying the cutoff frequency.
 - Optionally, r , a samplerate (by default 44100).
- `highpass($A, c \langle r \rangle$)` returns the input data after applying a simple first-order highpass RC filter.

Example.

Input:

```
f:=unapply(periodic(sign(x),x,-1/880,1/880),x);
s:=apply(f,soundsec(3));;
playsnd(highpass(createwav(s),5000))
```

Output:

The sound of the periodic signal after a simple first-order highpass RC filter has been applied.

15.2.11 Apply a moving average filter to a signal: `moving_average`

The `moving_average` command applies a moving average to a sample.

- `moving_average` takes two arguments:
 - A , an array of numeric values representing a sampled signal.
 - n , a positive integer n .
- `moving_average(A, n)` returns an array B obtained by applying a moving average filter of length n to A . The elements of B are defined by

$$B[i] = \frac{1}{n} \sum_{j=0}^{n-1} A[i+j]$$

for $i = 0, 1, \dots, L - n$, where L is the length of A .

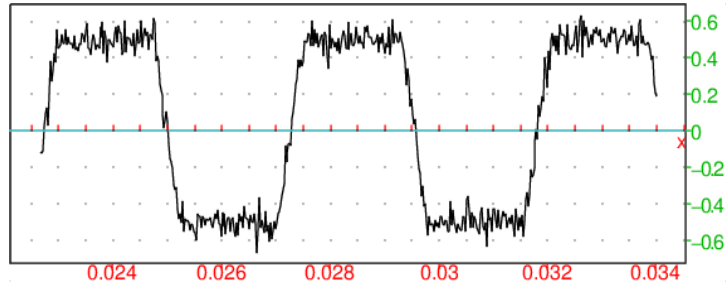
Moving average filters are fast and useful for smoothing time-encoded signals.

Examples.

- *Input:*

```
snd:=soundsec(2);
noise:=randvector(length(snd),normald,0,0.05);;
data:=0.5*threshold(3*sin(2*pi*220*snd),[-1.0,1.0])+noise;;
plotwav(createwav(data),range=[1000,1500])
```

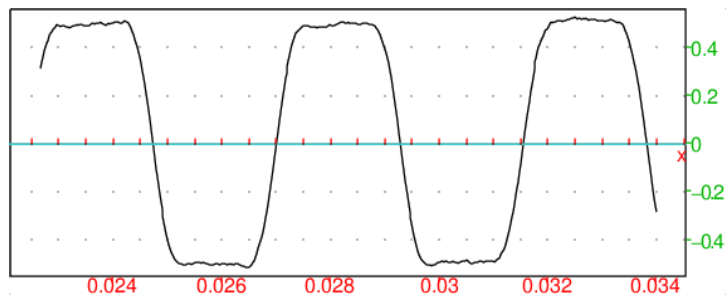
Output:



• *Input:*

```
fdata:=moving_average(data,25);
plotwav(createwav(fdata),range=[1000,1500])
```

Output:



15.2.12 Performing thresholding operations on an array: `threshold`

The `threshold` command changes data in an array by raising (or lower) the values to meet a given threshold.

- `threshold` takes two mandatory arguments and two optional arguments:
 - v , a vector of real or complex numbers.
 - a bound specification, which can be one of: This can be either:
 - * $b = b_0$ for real numbers b and b_0 .
 - * b , a real number (equivalent to $b = b$).
 - * $[l = l_0, u = u_0]$, for real numbers l, l_0, u, u_0 .
 - * $[l, u]$, a list of two real numbers (equivalent to $[l = l, u = u]$).
 - Optionally ' \prec ', a quoted comparison operator, one of ' $<$ ', ' \leq ', ' $>$ ', ' \geq ' (by default ' $<$ ').
 - Optionally, `abs=bool`, where `bool` is either `true` or `false` (by default `abs=false`). If `abs=true`, then the components of v must be real.

- `threshold(v, b = b_0 <'<')<')<')` returns the vector w whose k th component is:

$$w_k = \begin{cases} b_0, & v_k < b, \\ v_k, & \text{otherwise} \end{cases}$$

when v_k is real and

$$w_k = \begin{cases} b_0 \frac{v_k}{|v_k|}, & |v_k| < b, \\ v_k, & \text{otherwise} \end{cases}$$

when v_k is complex; for $k = 0, 1, \dots, \text{size}(v) - 1$

- `threshold(v, b = b_0 <,<'),abs=true)<,<')` returns the vector w whose k th component is:

$$w_k = \begin{cases} b_0, & |v_k| < b \text{ and } v_k > 0, \\ -b_0, & |v_k| < b \text{ and } v_k < 0, \\ v_k, & \text{otherwise} \end{cases}$$

- `threshold(v, [l = l_0, u = u_0] <,<')<,<')` returns the vector w whose k th component is:

$$w_k = \begin{cases} u_0, & u < v_k, \\ l_0, & v_k < l \\ v_k, & \text{otherwise} \end{cases}$$

when v_k is real and

$$w_k = \begin{cases} u_0 \frac{v_k}{|v_k|}, & u < |v_k|, \\ l_0 \frac{v_k}{|v_k|}, & |v_k| < l, \\ v_k, & \text{otherwise} \end{cases}$$

when v_k is complex; for $k = 0, 1, \dots, \text{size}(v) - 1$.

Examples.

- *Input:*

```
threshold([2,3,1,2,5,4,3,7],3)
```

Output:

```
[3,3,3,3,5,4,3,7]
```

- *Input:*

```
threshold([2,3,1,2,5,4,3,7],3=a,'>=')
```

Output:

```
[2,a,1,2,a,a,a,a]
```

- *Input:*

```
threshold([-2,-3,1,2,5,-4,3,-1],3=0,abs=true)
```

Output:

```
[0,-3,0,0,5,-4,3,0]
```

- *Input:*

```
threshold([-2,-3,1,2,5,-4,3,-1],3=0,'<=',abs=true)
```

Output:

```
[0,0,0,0,5,-4,0,0]
```

- *Input:*

```
threshold([-120,-11,-3,0,7,27,111,234],[-100,100])
```

Output:

```
[-100,-11,-3,0,7,27,100,100]
```

- *Input:*

```
threshold([-120,-11,-3,0,7,27,111,234],[-100=-inf,100=inf])
```

Output:

```
[-inf,-11,-3,0,7,27,+inf,+inf]
```

- In this example, a square-like wave is created from a single sine wave by clipping sample values.

Input:

```
data:=threshold(3*sin(2*pi*440*soundsec(2)),-1.0,1.0):;  
s:=createwav(data):;  
playsnd(s)
```

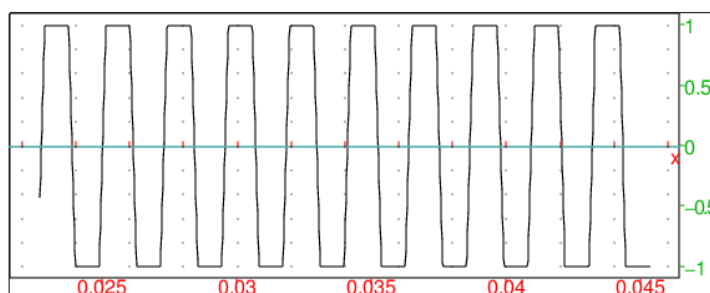
Output:

1

Input:

```
plotwav(s,range=[1000,2000])
```

Output:



15.2.13 Bartlett-Hann window function: bartlett_hann_window

The `bartlett_hann_window` command finds a Bartlett-Hahn window of a sequence, which can be used to examine a short segment when analyzing a long.

- `bartlett_hann_window` takes one mandatory argument and one optional argument:
 - v , a real vector with length n .
 - Optionally, an interval $n_0..n_1$ (by default $0..(n-1)$).
- `bartlett_hann_window($v \langle, n_1..n_2 \rangle$)` returns the elementwise product of $[v_{n_1}, \dots, v_{n_2}]$ and the vector w of length $N = n_2 - n_1 + 1$ defined by

$$w_k = a_0 + a_1 \left| \frac{k}{N-1} - \frac{1}{2} \right| - a_2 \cos \left(\frac{2k\pi}{N-1} \right)$$

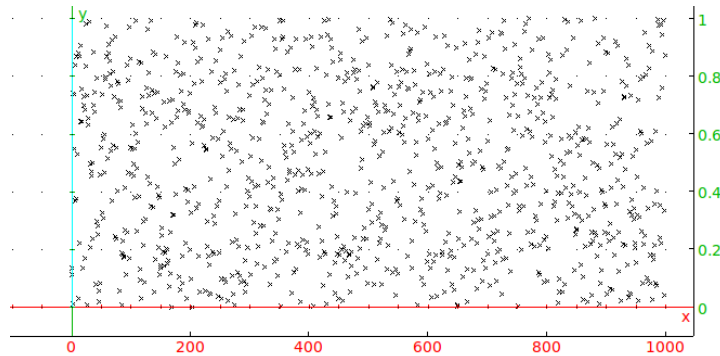
for $k = 0, 1, \dots, N-1$, where $a_0 = 0.62$, $a_1 = 0.48$ and $a_2 = 0.38$.

Example.

Input:

```
L0:=randvector(1000,0..1);;
scatterplot(L0);
```

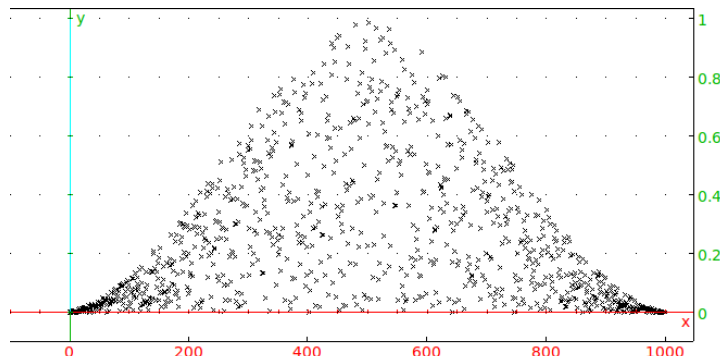
Output:



Input:

```
L:=bartlett_hann_window(L0);;
scatterplot(L);
```

Output:



15.2.14 Blackman-Harris window function: `blackman_harris_window`

The `blackman_harris_window` command finds a Blackman-Harris window of a sequence.

- `blackman_harris_window` takes one mandatory argument and one optional argument:
 - v , a real vector with length n .
 - Optionally, an interval $n_0..n_1$ (by default $0..(n-1)$).
- `blackman_harris_window(v⟨, n1..n2⟩)` returns the elementwise product of $[v_{n_1}, \dots, v_{n_2}]$ and the vector w of length $N = n_2 - n_1 + 1$ defined by

$$w_k = a_0 - a_1 \cos\left(\frac{2k\pi}{N-1}\right) + a_2 \cos\left(\frac{4k\pi}{N-1}\right) - a_3 \cos\left(\frac{6k\pi}{N-1}\right)$$

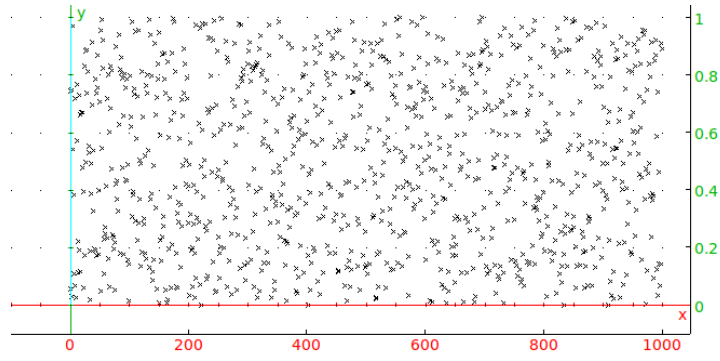
for $k = 0, 1, \dots, N-1$, where $a_0 = 0.35875$, $a_1 = 0.48829$, $a_2 = 0.14128$ and $a_3 = 0.01168$.

Example.

Input:

```
L0:=randvector(1000,0..1);;
scatterplot(L0);
```

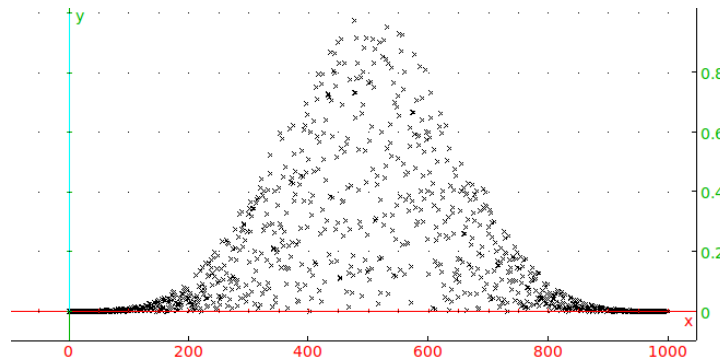
Output:



Input:

```
L:=blackman_harris_window(L0);;
scatterplot(L);
```

Output:



15.2.15 Blackman window function: blackman_window

The `blackman_window` command finds a Blackman window of a sequence.

- `blackman_window` takes one mandatory argument and two optional arguments:
 - v , a real vector with length n .
 - Optionally, α , a real number (by default 0.16).
 - Optionally, an interval $n_0..n_1$ (by default $0..(n-1)$).
- `blackman_harris_window($v \langle, \alpha, n_1..n_2 \rangle$)` returns the elementwise product of $[v_{n_1}, \dots, v_{n_2}]$ and the vector w of length $N = n_2 - n_1 + 1$ defined by

$$w_k = \frac{1 - \alpha}{2} - \frac{1}{2} \cos\left(\frac{2k\pi}{N-1}\right) + \frac{\alpha}{2} \cos\left(\frac{4k\pi}{N-1}\right)$$

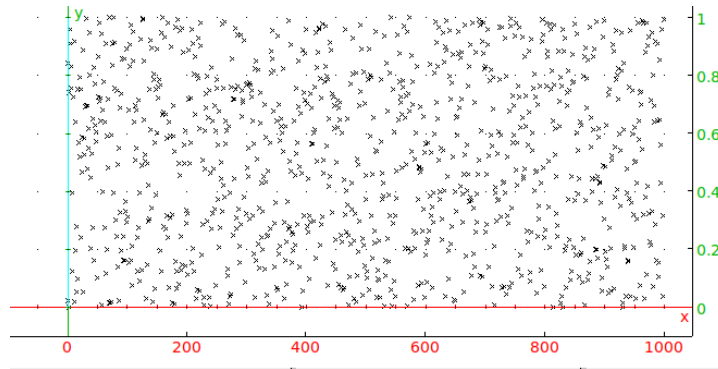
for $k = 0, 1, \dots, N-1$.

Example.

Input:

```
L0:=randvector(1000,0..1);;
scatterplot(L0);
```

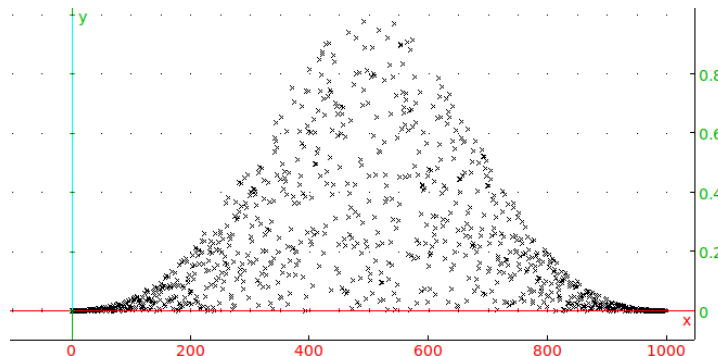
Output:



Input:

```
L:=blackman_window(L0);;
scatterplot(L);
```

Output:



15.2.16 Bohman window function: bohman_window

The `bohman_window` command finds a Bohman window of a sequence.

- `bohman_window` takes one mandatory argument and one optional argument:
 - v , a real vector with length n .
 - Optionally, an interval $n_0..n_1$ (by default $0..(n-1)$).
- `bohman_window($v \langle, n_1..n_2 \rangle$)` returns the elementwise product of $[v_{n_1}, \dots, v_{n_2}]$ and the vector w of length $N = n_2 - n_1 + 1$ defined by

$$w_k = (1 - x_k) \cos(\pi x_k) + \frac{1}{\pi} \sin(\pi x_k),$$

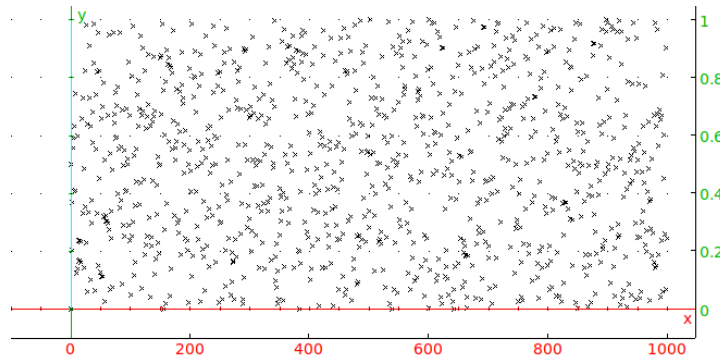
where $x_k = \left| \frac{2k}{N-1} - 1 \right|$ for $k = 0, 1, \dots, N-1$.

Example.

Input:

```
L0:=randvector(1000,0..1);;
scatterplot(L0);
```

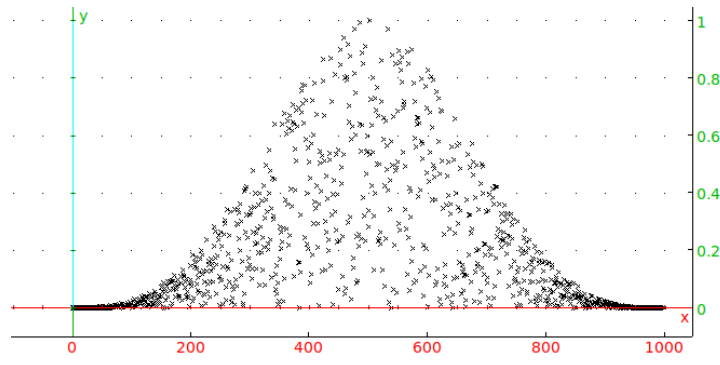
Output:



Input:

```
L:=bohman_window(L0);;
scatterplot(L);
```

Output:



15.2.17 Cosine window function: cosine_window

The `cosine_window` command finds a cosine window of a sequence.

- `cosine_window` takes one mandatory argument and two optional arguments:
 - v , a real vector with length n .
 - Optionally, α , a real number (by default 1).
 - Optionally, an interval $n_0..n_1$ (by default $0..(n-1)$).
- `cosine_window($v \langle \alpha, n_1..n_2 \rangle$)` returns the elementwise product of $[v_{n_1}, \dots, v_{n_2}]$ and the vector w of length $N = n_2 - n_1 + 1$ defined by

$$w_k = \sin^\alpha \left(\frac{k \pi}{N-1} \right)$$

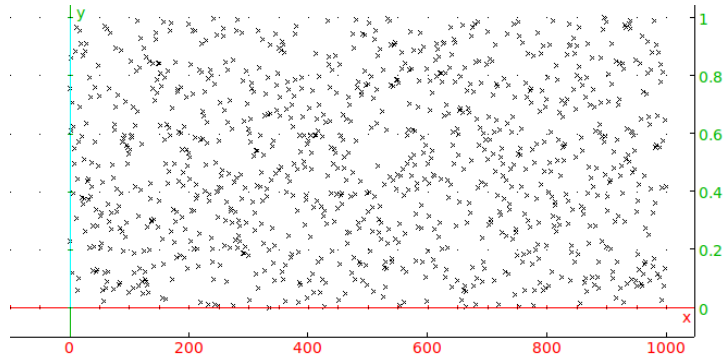
for $k = 0, 1, \dots, N-1$.

Example.

Input:

```
L0:=randvector(1000,0..1);;
scatterplot(L0);
```

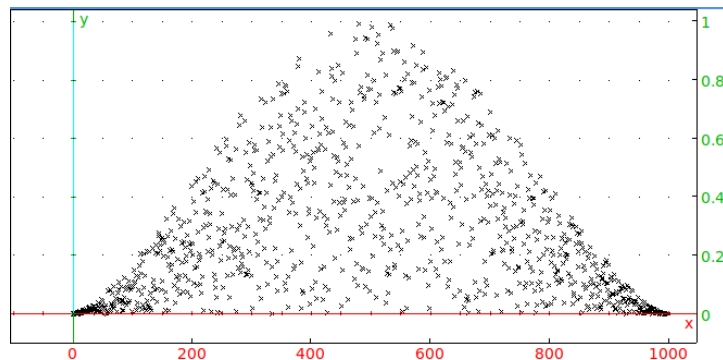
Output:



Input:

```
L:=cosine_window(L0,1.5);;
scatterplot(L);
```

Output:



15.2.18 Gaussian window function: `gaussian_window`

The `gaussian_window` command finds a Gaussian window of a sequence.

- `gaussian_window` takes one mandatory argument and two optional arguments:
 - v , a real vector with length n .
 - Optionally, α , a real number less than or equal to 0.5 (by default 0.1).
 - Optionally, an interval $n_0..n_1$ (by default $0..(n-1)$).
- `gaussian_window($v \langle \alpha, n_1..n_2 \rangle$)` returns the elementwise product of $[v_{n_1}, \dots, v_{n_2}]$ and the vector w of length $N = n_2 - n_1 + 1$ defined by

$$w_k = \exp \left(-\frac{1}{2} \left(\frac{k - (N-1)/2}{\alpha (N-1)/2} \right)^2 \right)$$

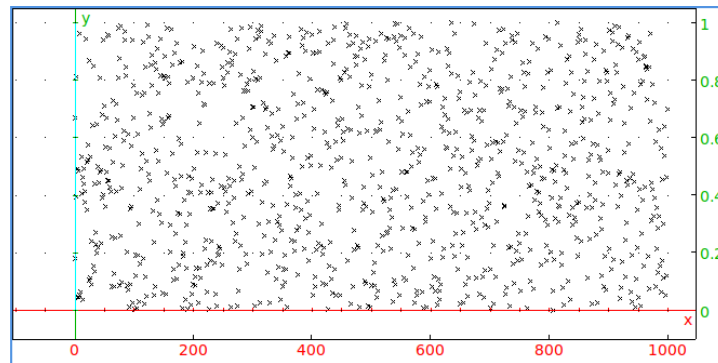
for $k = 0, 1, \dots, N-1$.

Example.

Input:

```
L0:=randvector(1000,0..1);;
scatterplot(L0);
```

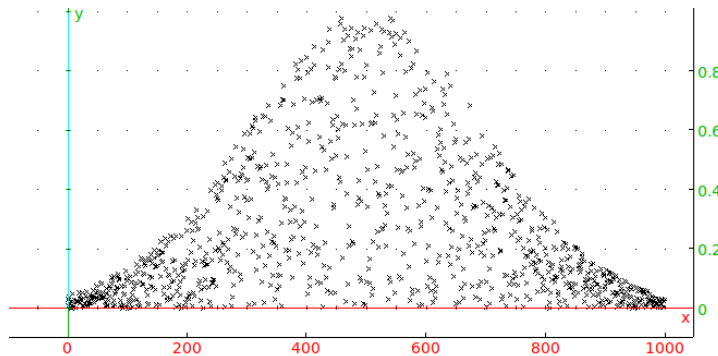
Output:



Input:

```
L:=gaussian_window(L0,0.4);;
scatterplot(L);
```

Output:



15.2.19 Hamming window function: `hamming_window`

The `hamming_window` command finds a Hamming window of a sequence.

- `hamming_window` takes one mandatory argument and one optional argument:
 - v , a real vector with length n .
 - Optionally, an interval $n_0..n_1$ (by default $0..(n-1)$).
- `hamming_window($v \langle n_1..n_2 \rangle$)` returns the elementwise product of $[v_{n_1}, \dots, v_{n_2}]$ and the vector w of length $N = n_2 - n_1 + 1$ defined by

$$w_k = \alpha - \beta \cos\left(\frac{2k\pi}{N-1}\right)$$

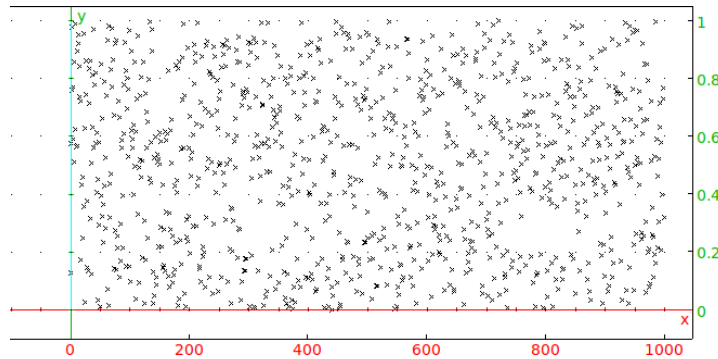
for $k = 0, 1, \dots, N-1$, where $\alpha = 0.54$ and $\beta = 1 - \alpha = 0.46$.

Example.

Input:

```
L0:=randvector(1000,0..1);;
scatterplot(L0);
```

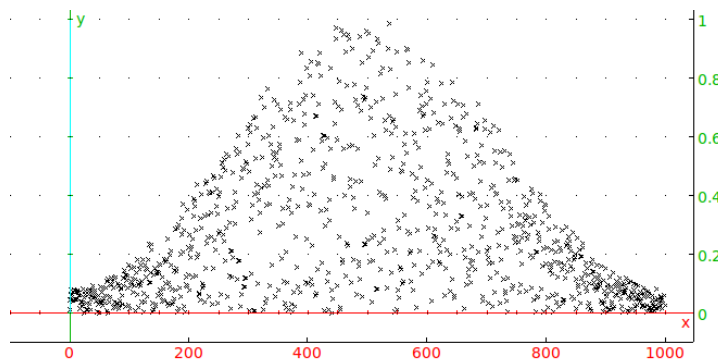
Output:



Input:

```
L:=hamming_window(L0);;
scatterplot(L);
```

Output:



15.2.20 Hann-Poisson window function: hann_poisson_window

The `hann_poisson_window` command finds a Hann-Poisson window of a sequence.

- `hann_poisson_window` takes one mandatory argument and two optional arguments:
 - v , a real vector with length n .
 - Optionally, α , a real number (by default 1).
 - Optionally, an interval $n_0..n_1$ (by default $0..(n-1)$).
- `hann_poisson_window($v \langle, \alpha, n_1..n_2 \rangle$)` returns the elementwise product of $[v_{n_1}, \dots, v_{n_2}]$ and the vector w of length $N = n_2 - n_1 + 1$ defined by

$$w_k = \frac{1}{2} \left(1 - \cos \frac{2k\pi}{N-1} \right) \exp \left(-\frac{\alpha |N-1-2k|}{N-1} \right)$$

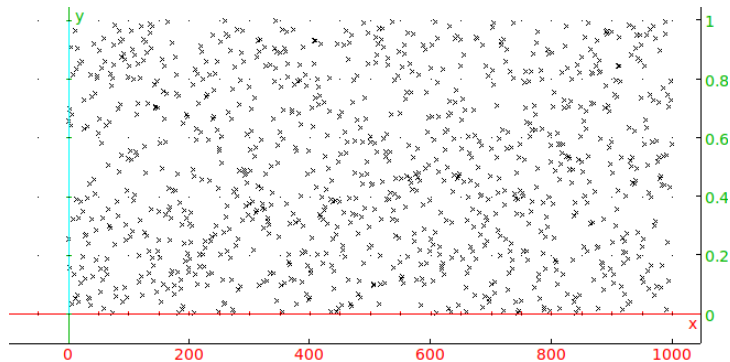
for $k = 0, 1, \dots, N-1$.

Example.

Input:

```
L0:=randvector(1000,0..1);;
scatterplot(L0);
```

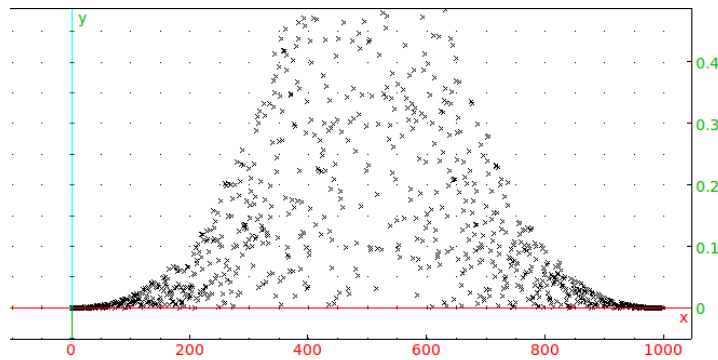
Output:



Input:

```
L:=hann_poisson_window(L0,2);;
scatterplot(L);
```

Output:



15.2.21 Hann window function: `hann_window`

The `hann_window` command finds a Hann window of a sequence.

- `hann_window` takes one mandatory argument and one optional argument:
 - v , a real vector with length n .
 - Optionally, an interval $n_0..n_1$ (by default $0..(n-1)$).
- `hann_window($v \langle n_1..n_2 \rangle$)` returns the elementwise product of $[v_{n_1}, \dots, v_{n_2}]$ and the vector w of length $N = n_2 - n_1 + 1$ defined by

$$w_k = \sin^2 \left(\frac{k \pi}{N-1} \right)$$

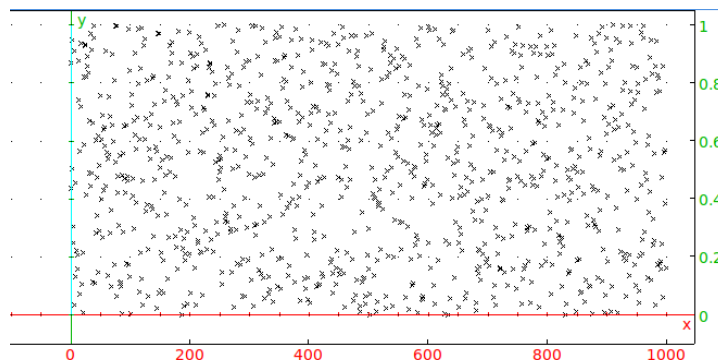
for $k = 0, 1, \dots, N-1$.

Example.

Input:

```
L0:=randvector(1000,0..1);;
scatterplot(L0);
```

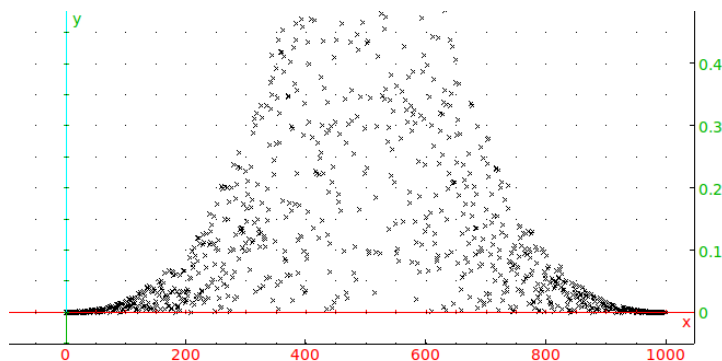
Output:



Input:

```
L:=hann_window(L0,2);;
scatterplot(L);
```

Output:



15.2.22 Parzen window function: `parzen_window`

The `parzen_window` command finds a Parzen window of a sequence.

- `parzen_window` takes one mandatory argument and one optional argument:
 - v , a real vector with length n .
 - Optionally, an interval $n_0..n_1$ (by default $0..(n-1)$).
- `parzen_window($v \langle n_1..n_2 \rangle$)` returns the elementwise product of $[v_{n_1}, \dots, v_{n_2}]$ and the vector w of length $N = n_2 - n_1 + 1$ defined by

$$w_k = \begin{cases} (1 - 6x_k^2(1 - x_k)), & \left| \frac{N-1}{2} - k \right| \leq \frac{N-1}{4}, \\ 2(1 - x_k)^3, & \text{otherwise,} \end{cases}$$

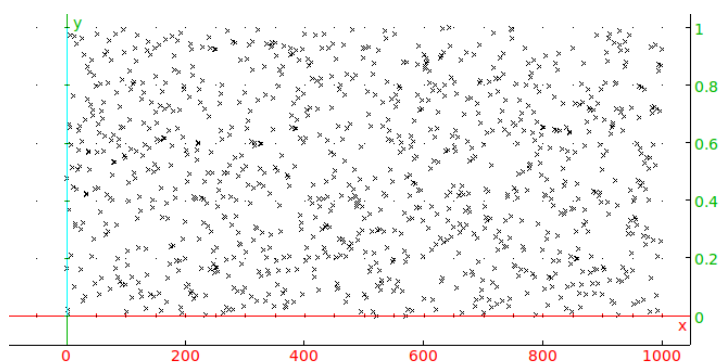
where $x_k = \left| 1 - \frac{2k}{N-1} \right|$ for $k = 0, 1, \dots, N-1$.

Example.

Input:

```
L0:=randvector(1000,0..1);
scatterplot(L0);
```

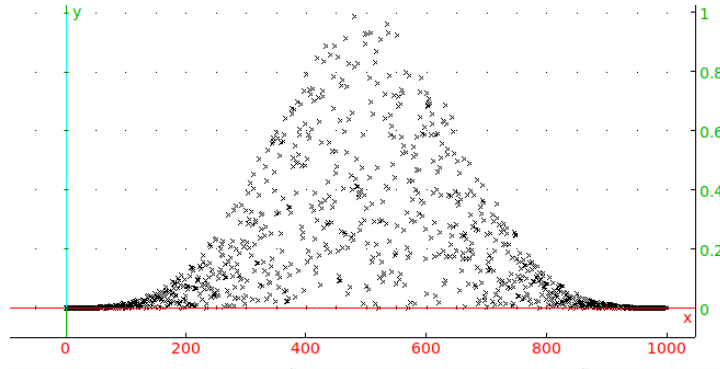
Output:



Input:

```
L:=parzen_window(L0);;
scatterplot(L);
```

Output:



15.2.23 Poisson window function: poisson_window

The `poisson_window` command finds a Poisson window of a sequence.

- `poisson_window` takes one mandatory argument and two optional arguments:
 - v , a real vector with length n .
 - Optionally, α , a real number (by default 1).
 - Optionally, an interval $n_0..n_1$ (by default $0..(n-1)$).
- `poisson_window($v \langle \alpha, n_1..n_2 \rangle$)` returns the elementwise product of $[v_{n_1}, \dots, v_{n_2}]$ and the vector w of length $N = n_2 - n_1 + 1$ defined by

$$w_k = \exp \left(-\alpha \left| \frac{2k}{N-1} - 1 \right| \right)$$

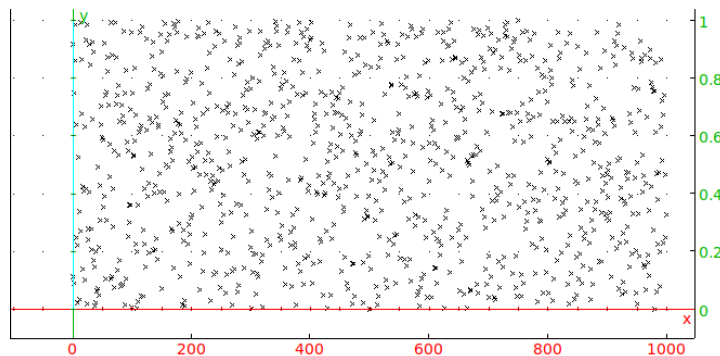
for $k = 0, 1, \dots, N-1$.

Example.

Input:

```
L0:=randvector(1000,0..1);;
scatterplot(L0);
```

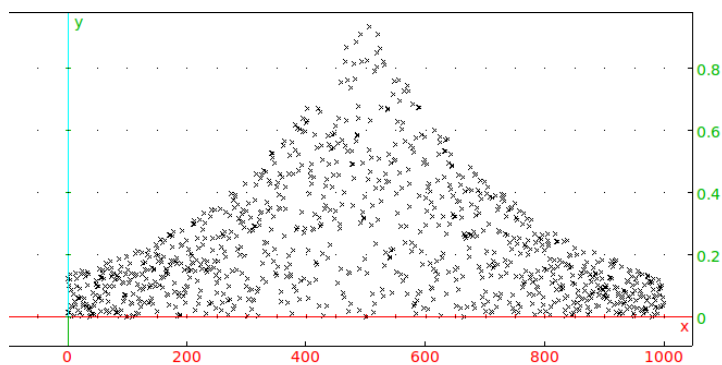
Output:



Input:

```
L:=poisson_window(L0,2)::  
scatterplot(L);
```

Output:



15.2.24 Riemann window function: `riemann_window`

The `riemann_window` command finds a Riemann window of a sequence.

- `riemann_window` takes one mandatory argument and one optional argument:
 - v , a real vector with length n .
 - Optionally, an interval $n_0..n_1$ (by default $0..(n-1)$).
- `riemann_window($v \langle n_1..n_2 \rangle$)` returns the elementwise product of $[v_{n_1}, \dots, v_{n_2}]$ and the vector w of length $N = n_2 - n_1 + 1$ defined by

$$w_k = \begin{cases} 1, & k = \frac{N-1}{2}, \\ \frac{\sin(\pi x_k)}{\pi x_k}, & \text{otherwise,} \end{cases}$$

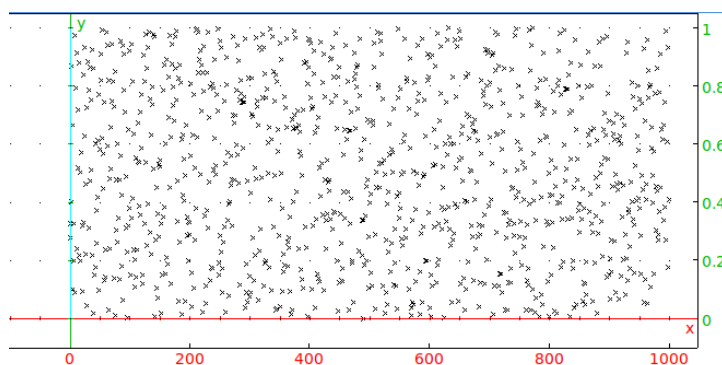
where $x_k = \frac{2k}{N-1} - 1$ for $k = 0, 1, \dots, N-1$.

Example.

Input:

```
L0:=randvector(1000,0..1)::  
scatterplot(L0);
```

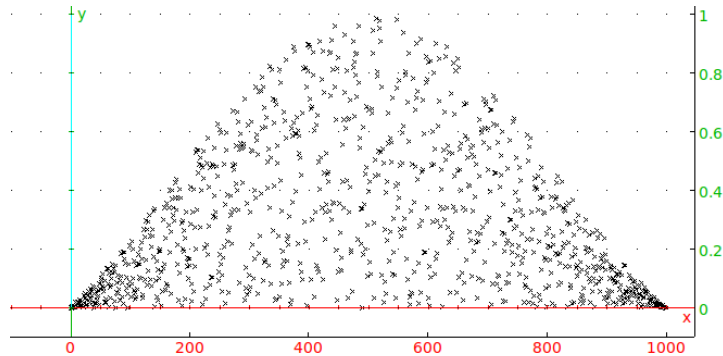
Output:



Input:

```
L:=riemann_window(L0);;
scatterplot(L);
```

Output:



15.2.25 Triangular window function: `triangle_window`

The `triangle_window` command finds a triangle window of a sequence.

- `triangle_window` takes one mandatory argument and two optional arguments:
 - v , a real vector with length n .
 - Optionally, d , either -1, 0 or 1 (by default 0).
 - Optionally, an interval $n_0..n_1$ (by default $0..(n-1)$).
- `triangle_window($v \langle, d, n_1..n_2 \rangle$)` returns the elementwise product of $[v_{n_1}, \dots, v_{n_2}]$ and the vector w of length $N = n_2 - n_1 + 1$ defined by

$$w_k = 1 - \left| \frac{n - \frac{N-1}{2}}{\frac{N+d}{2}} \right|$$

for $k = 0, 1, \dots, N - 1$.

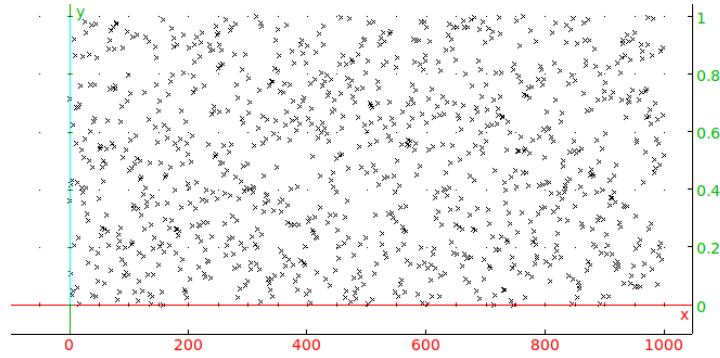
The case $d = -1$ is called the Bartlett window function.

Example.

Input:

```
L0:=randvector(1000,0..1);;
scatterplot(L0);
```

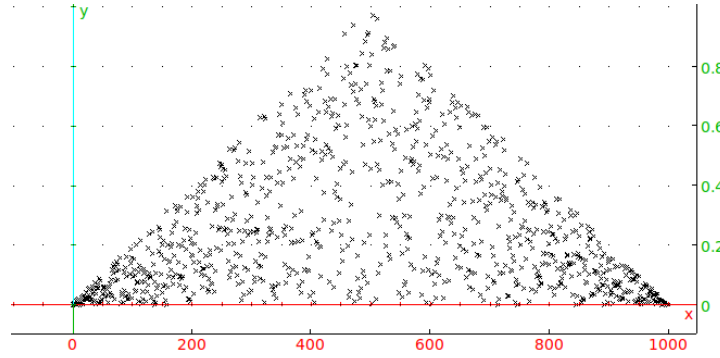
Output:



Input:

```
L:=triangle_window(L0,1);;
scatterplot(L);
```

Output:



15.2.26 Tukey window function: `tukey_window`

The `tukey_window` command finds a Tukey window of a sequence.

- `tukey_window` takes one mandatory argument and two optional arguments:
 - v , a real vector with length n .
 - Optionally, α , a real number in $[0, 1]$ (by default 0.5).
 - Optionally, an interval $n_0..n_1$ (by default $0..(n-1)$).
- `tukey_window($v \langle \alpha, n_1..n_2 \rangle$)` returns the elementwise product of $[v_{n_1}, \dots, v_{n_2}]$ and the vector w of length $N = n_2 - n_1 + 1$ defined by

$$w_k = \begin{cases} \frac{1}{2} \left(1 + \cos \left(\pi \left(\frac{k}{\beta} - 1 \right) \right) \right), & k < \beta, \\ 1, & \beta \leq k \leq (N-1) \left(1 - \frac{\alpha}{2} \right), \\ \frac{1}{2} \left(1 + \cos \left(\pi \left(\frac{k}{\beta} - \frac{2}{\alpha} + 1 \right) \right) \right), & \text{otherwise,} \end{cases}$$

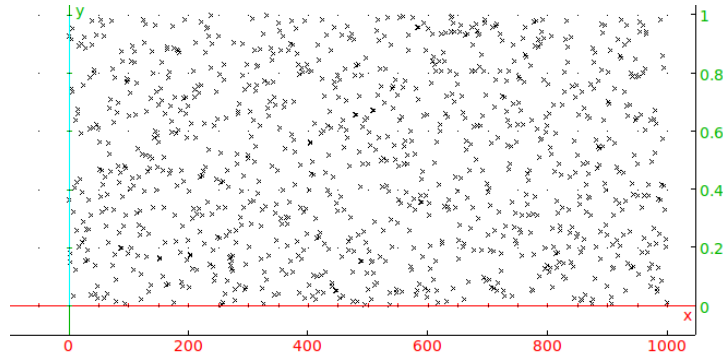
where $\beta = \frac{\alpha(N-1)}{2}$, for $k = 0, 1, \dots, N-1$.

Example.

Input:


```
L0:=randvector(1000,0..1);;
scatterplot(L0);
```

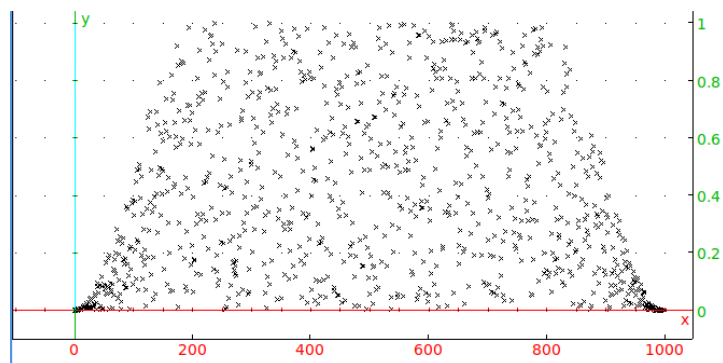
Output:



Input:

```
L:=tukey_window(L0,0.4);;
scatterplot(L);
```

Output:



15.2.27 Welch window function: `welch_window`

The `welch_window` command finds a Welch window of a sequence.

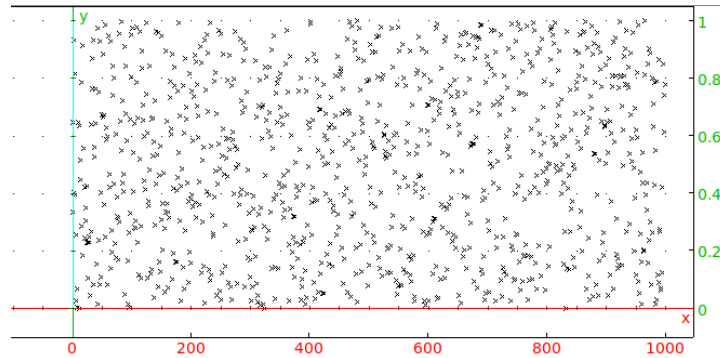
- `welch_window` takes one mandatory argument and one optional argument:
 - v , a real vector with length n .
 - Optionally, an interval $n_0..n_1$ (by default $0..(n-1)$).
- `welch_window($v \langle n_1..n_2 \rangle$)` returns the elementwise product of $[v_{n_1}, \dots, v_{n_2}]$ and the vector w of length $N = n_2 - n_1 + 1$ defined by

$$w_k = 1 - \left(\frac{k - \frac{N-1}{2}}{\frac{N-1}{2}} \right)^2$$

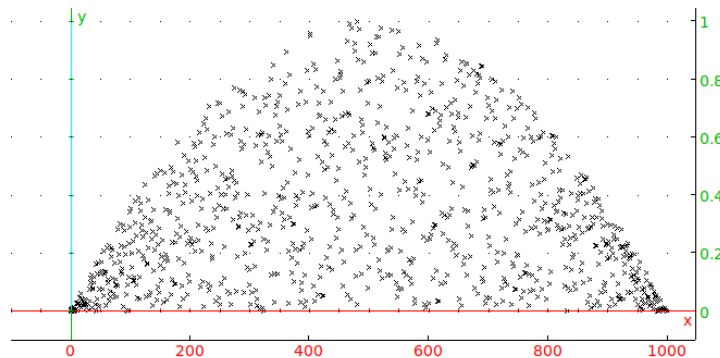
for $k = 0, 1, \dots, N-1$.

Example.*Input:*

```
L0:=randvector(1000,0..1);;
scatterplot(L0);
```

Output:*Input:*

```
L:=welch_window(L0);;
scatterplot(L);
```

Output:**15.2.28 An example: static noise removal by spectral subtraction**

In this section, you use Xcas to implement a simple algorithm for static noise removal based on the spectral subtraction method. For a theoretical overview see the paper "Noise Reduction Based on Modified Spectral Subtraction Method" by Ekaterina Verteletskaya and Boris Simak (2011), *International Journal of Computer Science*, 38:1 ([PDF](#)).

Efficiency of the spectral subtraction method largely depends on a good noise spectrum estimate. Below is the code for a function `noiseprof` that takes `data` and `wlen` as its arguments. These are, respectively, a signal chunk containing only noise and the window length for signal segmentation (the best values are powers of two, such as 256, 512 or 1024). The function returns an estimate of the noise power spectrum obtained by averaging the power spectra of a (not too large) number of distinct chunks of `data` of length `wlen`. The Hamming window function is applied prior to FFT.

```

noiseprof(data,wlen):={
    local N,h,dx,x,v,cnt;
    N:=length(data);
    h:=wlen/2;
    dx:=min(h,max(1,(N-wlen)/100));
    v:=[0$wlen];
    cnt:=0;
    for (x:=h;x<N-h;x+=dx) \{
        v+=abs(fft(hamming_window(
            mid(data,floor(x)-h,wlen)))).^2;
        cnt++;
    };
    return 1.0/cnt*v;
};

```

The main function is `noisered`, which takes three arguments: the input signal `data`, the noise power spectrum `np` and the "spectral floor" parameter `beta` (β , the minimum power level). The function performs subtraction of the noise spectrum in chunks of length `wlen` (the length of list `np`) using the overlap-and-add approach with Hamming window function. For details see Section 3A of the paper "Speech Enhancement using Spectral Subtraction-type Algorithms: A Comparison and Simulation Study" by Navneet Upadhyay and Abhijit Karmakar (2015), *Procedia Computer Science*, vol. 54, pp. 574–584 ([PDF](#)).

```

noisered(data,np,beta):={
    local wlen,h,N,L,padded,out,j,k,s,ds,r,alpha;
    wlen:=length(np);
    N:=length(data);
    h:=wlen/2;
    L:=0;
    repeat L+=wlen; until L>=N;
    padded:=concat(data,[0$(L-N)]);
    out:=[0$L];
    for (k:=0;k<L-wlen;k+=h) {
        s:=fft(hamming_window(mid(padded,k,wlen)));
        alpha:=max(1,4-3*sum(abs(s).^2)/(20*sum(np)));
        r:=ifft(zip(max,abs(s).^2-alpha*np,beta*np).^(1/2)
            .*exp(i*arg(s))));
        for (j:=0;j<wlen;j++) {
            out[k+j]+=re(r[j]);
        };
    };
    return mid(out,0,N);
};

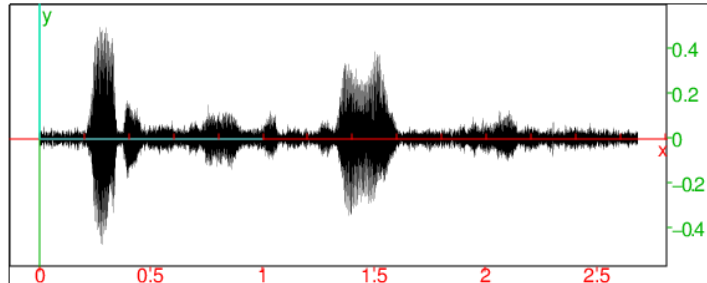
```

To demonstrate the efficiency of the algorithm, you can test it on a small speech sample with an audible amount of static noise. Assume that the corresponding `wav` file `noised.wav` is stored in the directory `sounds`.

Input:

```
clip:=readwav("/path/to/sounds/noised.wav");; plotwav(clip)
```

Output:



Speech starts after approximately 0.2 seconds of pure noise. You can use that part of the clip for obtaining an estimate of the noise power spectrum with `wlen` set to 256.

Input:

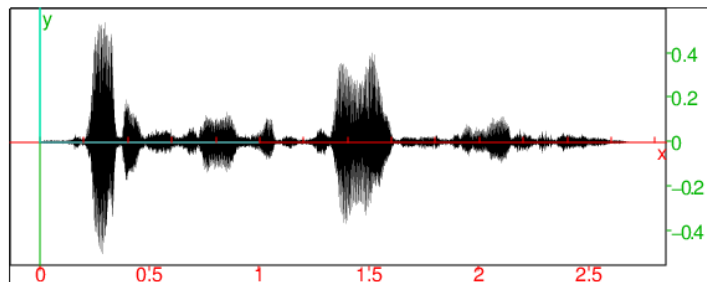
```
noise:=channel_data(clip,range=0.0..0.15);;
np:=noiseprof(noise,256);;
```

Now call the `noisered` function with $\beta = 0.03$:

Input:

```
c:=noisered(channel_data(clip),np,0.03);;
cleaned:=createwav(c);; plotwav(cleaned)
```

Output:



You can clearly see that the noise level is significantly lower than in the original clip. One can also use the `playsnd` command to compare the input with the output by hearing, which reveals that the noise is still present but in a lesser degree (the parameter β controls how much noise is "left in").

The algorithm implemented in this section is not particularly fast (removing the noise from a two and a half seconds long recording took 20 seconds of computation time), but serves as a proof of concept and demonstrates the efficiency of noise removal.

15.3 Images

15.3.1 Image structure in Xcas

An image in `Xcas` is a list with the following elements.

- The first element is itself a list of three integers; the number of channels (which will be 3 or 4), the number *n* of rows and the number *p* of columns used for the dimension of the image. Each channel will be an *n*×*p* matrix of integers between 0 and 255.
- A red channel.
- A green channel.
- A transparency channel.
- A blue channel.

The color of the point at line *i* and column *j* is determined by the values of the *i,j*th entry of the matrices.

Note that the number of points in the structure isn't necessarily the same as the number of pixels on the screen when it is drawn. It is possible that a single point in the structure is represented by a small rectangle of pixels when it is displayed on the screen.

15.3.2 Reading images: readrgb

The `readrgb` command reads an `Xcas` image structure (see Section 15.3.1 p.1108).

- `readrgb` command takes one argument:
filename, the name of an image file (it can be `.jpg`, `.png` or `.gif`).
- `readrgb(filename)` returns an `Xcas` image structure for the image in *filename*.

15.3.3 Viewing images

`Xcas` can display images in rectangles in two-dimensions or on surfaces in three-dimensions with the `gl_texture` property of the object (see Section 8.3.1 p.663).

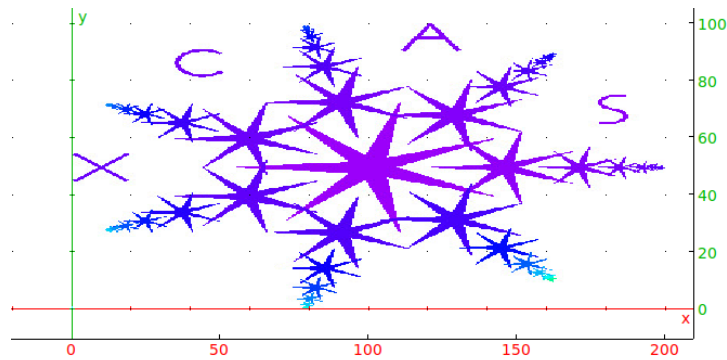
Examples.

Assume that `xcaslogo.png` is a picture of the `Xcas` logo.

- *Input:*

```
rectangle(0,200,1/2,gl_texture="xcaslogo.png")
```

Output:

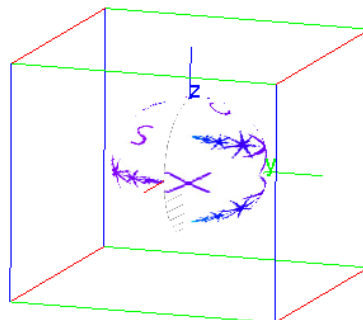


- *Input:*

```
sphere([0,0,0],1,gl_material=[gl_texture,"xcaslogo.png"])
```

- *Output:*

mouse plan $0.916x+0.322y+0.24z=0$



15.3.4 Creating or recreating images: writergb

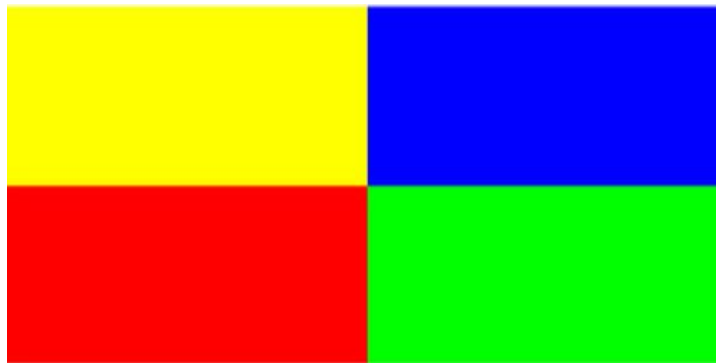
The `writergb` command writes images to `png` files; the image can be given by the `Xcas` image structure (see Section 15.3.1 p.1108, this is what is read in with `readrgb`) or a simplified version of this structure.

To write an image given by the `Xcas` image structure to a file:

- `writergb` takes two arguments:
 - *filename*, a file name.
 - *image*, an image in `Xcas` format.
- `writergb(filename,image)` writes the image *image* to the file *filename*.

Examples.

- Assume the the following image is stored in file `image1234.jpg`.



After reading it into a variable name with `readrgb`:

Input:

```
a:= readrgb("image1234.jpg")
```

the variable `a` will contain a list,

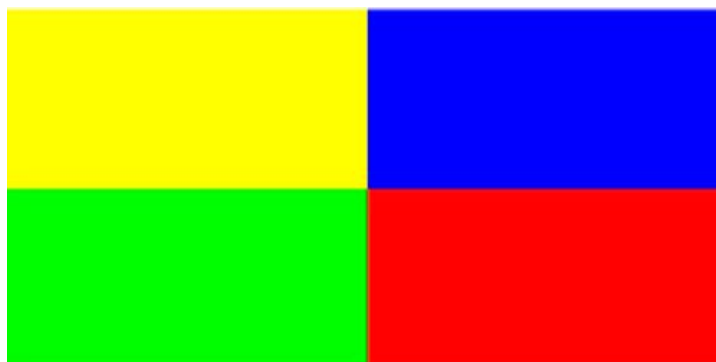
- `a[0]` will be `[4,250,500]`, the number of channels, the height and the width of the image.
- `a[1]`, the red channel,
- `a[2]`, the green channel,
- `a[3]`, the transparency channel,
- `a[4]`, the blue channel.

Then:

Input:

```
writergb("image2134.png",[a[0],a[2],a[1],a[3],a[4]])
```

Output:



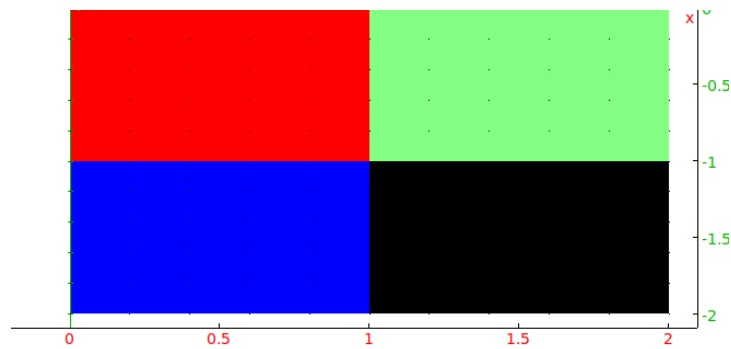
and the image file `image2134.png` will be created. This image is simply `image1234.png` with the green and red colors switched.

- For simple cases, you can type the `Xcas` image format in by hand.

Input:

```
writergb("image1.png", [[4,2,2],[[255,0],[0,0]],[[0,255],[0,0]],[[255,125],[255,255]],[[0,0],[255,0]]])
```

Output:



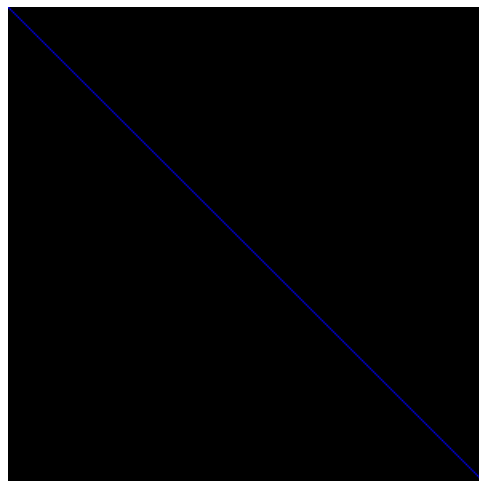
The transparency value of 125 for the upper right point makes it partially transparent and mutes the color.

- For larger images, in some cases the matrix operations of **Xcas** can be used to create the channels.

Input:

```
writergb("image2.png", [[4,300,300],makemat(0,300,300),makemat(0,300,300),makemat(255,300,300),makemat(0,300,300)+idn(300)*255])
```

Output:



The simplified version of the **Xcas** image description doesn't involve stating the number of channels, the size of the image, or the transparency. There is a full color version of this simplified form and a grayscale version.

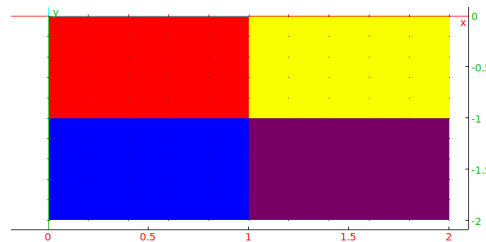
To create a full color image using the simple description:

- **writergb** command takes four arguments:

- `filename`, the name of the file to store the image.
 - R , a matrix for the red channel.
 - G , a matrix for the green channel.
 - B , a matrix for the blue channel.
- `writergb(filename, R, G, B)` draws the image given by the matrices to the file `filename`.

Example.*Input:*

```
writergb("image2.png", [[255,250],[0,120]], [[0,255],[0,0]], [[0,0],[255,100]])
```

Output:

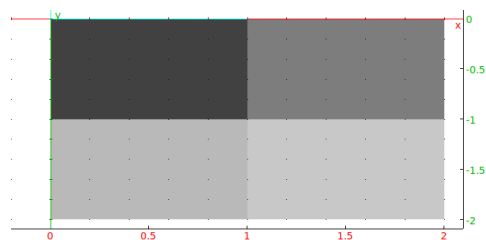
This image will be in the file `image2.png`.

To create a grayscale image using the simple description:

- `writergb` command takes two arguments:
 - `filename`, the name of the file to store the image.
 - M , a matrix representing how dark each point is (where 0 is black and 255 is white).
- `writergb(filename, M)` draws the image given by M to the file `filename`.

Example.*Input:*

```
writergb("image3.png", [[65,125],[185,200]])
```

Output:

This image will be in the file `image3.png`.

Chapter 16

Using giac inside a program

16.1 Using giac inside a C++ program

To use `giac` inside of a C++ program, put

```
#include <giac/giac.h>
```

at the beginning of the file. To compile the file, use

```
c++ -g progname.cc -lgiac -lgmp
```

After compiling, there will be a file `a.out` which can be run with the command

```
./a.out
```

For example, put the following program in a file named `pgcd.cc`.

```
// -*- compile-command: "g++ -g pgcd.cc -lgiac -lgmp" -*-
#include <giac/config.h>
#include <giac/giac.h>

using namespace std; using namespace giac;

gen pgcd(gen a,gen b){ gen q,r; for (;b!=0;){ r=irem(a,b,q); a=b; b=r;
} return a; }

int main(){ cout << "Enter 2 integers "; gen a,b; cin >> a >> b; cout
<< pgcd(a,b) << endl; return 0; }
```

After compiling this with

```
c++ -g pgcd.cc -lgiac -lgmp
```

and running it with

```
./a.out
```

there will be a prompt

```
Enter 2 integers
```

After entering two integers, such as with

```
Enter 2 integers 30 36
```

the result will appear:

```
6
```

16.2 Defining new giac functions

New `giac` functions can be defined with a C++ program. All data in the program used in formal calculations needs to be `gen` type. A variable `g` can be declared to be `gen` type with

```
gen g;
```

In this case, `g.type` can have different values.

- If `g.val` is an integer type `int`, then `g.type` will be `_INT_`.
- If `g._DOUBLE_val` is a real double, `g.type` will be `_DOUBLE_`.
- If `g._SYMBptr` is type `symbolic`, then `g.type` will be `_SYMB`.
- If `g._VECTptr` is a vector, type `vector`, then `g.type` will be `_VECT`.
- If `g._ZINTptr` is an integer type `zint`, then `g.type` will be `_ZINT`.
- If `g._IDNTptr` is an identifier, type `idnt`, then `g.type` will be `_IDNT`.
- If `g._CPLXptr` is a complex type `complex`, then `g.type` will be `_CPLX`.

As an example, put the following program in a file called `pgcd.cpp`.

```
// -*- mode:C++ ; compile-command: "g++ -I.. -fPIC -DPIC -g -c pgcd.cpp -o pgcd.o"
// ln -sf pgcd.lo pgcd.o && \
// gcc -shared pgcd.lo -lc -lgiac -Wl,-soname -Wl,libpgcd.so.0 -o \
// libpgcd.so.0.0.0 && ln -sf libpgcd.so.0.0.0 libpgcd.so.0 && \
// ln -sf libpgcd.so.0.0.0 libpgcd.so" -*-
using namespace std;
#include <stdexcept>
#include <cmath>
#include <cstdlib>
#include <giac/config.h>
#include <giac/giac.h>
//#include "pgcd.h"

#ifdef NO_NAMESPACE_GIAC namespace giac { #endif // undef
NO_NAMESPACE_GIAC

gen monpgcd(const gen & a0,const gen & b0){ gen q,r,a=a0,b=b0; for
(;b!=0;){ r=irem(a,b,q); a=b; b=r; } return a; } gen _monpgcd(const
gen & args,GIAC_CONTEXT){ if ( (args.type!=_VECT) ||
```

```

    (args._VECTptr->size()!=2)) setsizeerr(); vecteur &v=*args._VECTptr;
    return monpgcd(v[0],v[1]); } const string _monpgcd_s("monpgcd");
    unary_function_eval __monpgcd(0,&_monpgcd,_monpgcd_s);
    unary_function_ptr at_monpgcd (&__monpgcd,0,true);

#ifdef NO_NAMESPACE_GIAC } // namespace giac #endif // undef
NO_NAMESPACE_GIAC

```

After compiling this with the commands after the `compile-command` in the header, namely

```

g++ -I.. -fPIC -DPIC -g -c pgcd.cpp -o pgcd.lo && \
    ln -sf pgcd.lo pgcd.o && \
    gcc -shared pgcd.lo -lc -lgiac -Wl,-soname -Wl,libpgcd.so.0 -o \
    libpgcd.so.0.0.0 && ln -sf libpgcd.so.0.0.0 libpgcd.so.0 && \
    ln -sf libpgcd.so.0.0.0 libpgcd.so

```

the new command can be inserted with the `insmod` command in `giac`, where `insmod` takes the full absolute path of the `libpgcd.so` file as argument.

Input:

```
insmod("/path/to/file/libpgcd.so")
```

Afterwards, the `monpgcd` command will be another `giac` command.

Input:

```
monpgcd(30,36)
```

Output: