# Symbolic algebra and Mathematics with `Xcas`

Renée De Graeve, Bernard Parisse
University of Grenoble I

# Contents

Chapter 1

# Index

# Index

# Chapter 2

# The CAS functions

## 2.1 Symbolic constants : `e pi infinity i`

`e` is the number $\exp(1)$;
`pi` is the number $\pi$.
`infinity` is unsigned $\infty$.
`+infinity` is $+\infty$.
`-infinity` is $-\infty$.
`i` is the complex number $i$.

## 2.2 Booleans

### 2.2.1 The values of a boolean : `true false`

The value of a boolean is `true` or `false`.
The synonyms are :
`true` or `TRUE` or `1`,
`false` or `FALSE` or `0`.
Tests or conditions are boolean functions.

### 2.2.2 Tests : `==, !=, >, >=, <, =<`

`==, !=, >, >=, <, =<` are infixed operators.
`a==b` tests the equality between `a` and `b` and returns `1` if `a` is equal to `b` and `0` otherwise.
`a!=b` returns `1` if `a` and `b` are different and `0` otherwise.
`a>=b` returns `1` if `a` is superior or equal to `b` and `0` otherwise.
`a>b` returns `1` if `a` is strictly superior to `b` and `0` otherwise.
`a<=b` returns `1` if `a` is inferior or equal to `b` and `0` otherwise.
`a<b` returns `1` if `a` is strictly inferior to `b` and `0` otherwise.
To write an algebraic function having the same result as an `if...then...else`, we use the boolean function `ifte`.
For example :

```
f(x):=ifte(x>0,true,false)
```

defines the boolean function $f$ such that `f(x)= true` if $x \in ]0;+\infty[$ and `f(x)=false` if $x \in ]-\infty;0]$.
Input :

```
f(0)==0
```

Output :

```
1
```

**Look out !**
`a=b` is not a boolean !!!!
`a==b` is a boolean.

### 2.2.3 Boolean operators : `or xor and not`

`or` (or `||`), `xor`, `and` (or `&&`) are infixed operators.
`not` is a prefixed operators.
If `a` and `b` are two booleans :
`(a or b) (a || b)` returns `0` (or `false`) if `a` and `b` are equal to 0 and returns `1` (or `true`) otherwise.
`(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 equal to 0 or if `a` and `b` are equal to 1 (it is the "exclusive or").
`(a and b)` or `(a && b)` returns `1` (or `true`) if `a` and `b` are equal to 1 and `0` (or `false`) otherwise.
`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`).
Input :

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

Output :

```
1
```

Input :

```
1>=0 xor 1>0
```

Output :

```
0
```

Input :

```
1>=0 and 1>0
```

Output :

```
1
```

Input :

```
not(0==0)
```

Output :

```
0
```

### 2.2.4 Transform a boolean expression as a list : `exp2list`

`exp2list` returns the list `[expr0,expr1]` when the argument is `(var=expr0) or (var=expr1)`.
`exp2list` is used in TI mode for easier processing of the answer of a `solve` command.
Input :

```
exp2list((x=2) or (x=0))
```

Output :

```
[2,0]
```

Input :

```
exp2list((x>0) or (x<2))
```

Output :

```
[0,2]
```

In TI mode input :

```
exp2list(solve((x-1)*(x-2)))
```

Output :

```
[1,2]
```

### 2.2.5 Evaluate booleans : `evalb`

Inside Maple, `evalb` evaluates an boolean expression. Since `Xcas` evaluates booleans automatically, `evalb` is only here for compatibility and is equivalent to `eval`
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
```

## 2.3 Operators bit to bit

### 2.3.1 Operators `bitor, bitxor, bitand`

The integers may be written using hexadecimal notation 0x... for example 0x1f represents 16+15=31 in decimal writting. Integers may also be outputted in hexadecimal notation (click on the red CAS status button and select `Base (Integers)`).
`bitor` is the logical inclusive `or` (bit to bit).
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 the logical exclusive `or` (bit to bit).
Input :

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

or input :

```
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` en base 2 and so, is equal to `42`.

`bitand` is the logical `and` (bit to bit).
Input :

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

or input :

```
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`.

### 2.3.2 Hamming distance bit to bit : `hamdist`

The Hamming distance is the number of differences of the bits of the two arguments.
Input :

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

or input

```
ha
mdist(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`.

## 2.4 Strings

### 2.4.1 Character and string : `"`

`"` is used to delimit a string. A character is a string of length one.
Do not confuse `"` with `'` (or `quote`) which is used to avoid evaluation of an expression . For example, `"a"` returns a string of one character but `'a'` or `quote(a)` returns the variable `a` unevaluated.

When a string is input in a command line, it is evaluated to itself hence the output is the same string. Use + to concatenate two strings or a string and another object.
Example :
Input :

```
"Hello"
```

`"Hello"` is the input and also the output.
Input :

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

Output :

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

The index notation is used to get the n-th character of a string, (as for lists). Indices begin at 0 in Xcas mode, 1 in other modes.
Example :
Input :

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

Output :

```
"e"
```

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

- `head(s)` returns the first character of the string `s`.
  Input :

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

  Output :

  ```
  "H"
  ```

- `mid(s,p,q)` returns the part of size `q` of the string `s` begining at the character of index `p`.
  Remember that the first index is 0 in Xcas mode.
  Input :

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

  Output :

  ```
  "ell"
  ```

- `tail(s)` returns the string `s` without its first character.
  Input :

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

  Output :

  ```
  "ello"
  ```

### 2.4.3 Concatenation of a sequence of words : `cumSum`

`cumSum` works on strings like on expressions by doing partial concatenation.
`cumSum` takes as argument a list of strings.
`cumSum` returns a list of strings where the element of index $k$ is the concatenation of the string of index $k$, with the strings of indices 0 to $k-1$ .
Input :

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

Output :

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

### 2.4.4 ASCII code of a character : `ord`

`ord` takes as argument a string `s` (resp. a list `l` of strings).
`ord` returns the ASCII code of the first character of `s` (resp the list of the ASCII codes of the first character of the elements of `l`).
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]
```

### 2.4.5 ASCII code of a string : `asc`

`asc` takes as argument a string `s`. `asc` returns the list of the ASCII codes of the characters of `s`.
Input :

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

Output :

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

Input :

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

Output :

```
[97]
```

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

`char` takes as argument a list `l` of ASCII codes.
`char` returns the string whose characters have as ASCII codes the elements of the list `l`.
Input :

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

Output :

```
"abcd"
```

Input :

```
char(97)
```

Output :

```
"a"
```

Input :

```
char(353)
```

Output :

```
"a"
```

En effet 353-256=97.

### 2.4.7 Find a character in a string : `inString`

`inString` takes two arguments : a string `S` and a character `c`.
`inString` tests if the character `c` is in the string `S`.
`inString` returns `"the index of its first occurence"` or `-1` if `c` is not in `S`.
Input :

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

Output :

```
3
```

Input :

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

Output :

```
-1
```

### 2.4.8 Concat objects into a string : `cat`

`cat` takes as argument a sequence of objects.
`cat` concat these objects into a string.
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"
```

### 2.4.9 Add an object to a string : +

+ is an infixed operator (resp `'+'` is a prefixed operator.
If + (resp `'+'`) takes as argument a string (resp. a sequence of objects with a string as first or second argument), the result is the concatenation of these objects into a string.
**warning**
+ is infixed and `'+'` is prefixed.
Input :

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

Ou on tape :

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

Output :

```
"abcd3d"
```

Input :

```
c:=5
```

Then input:

```
"abcd"+c+"e"
```

Or input:

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

Output :

```
"abcd5e"
```

### 2.4.10 Transform an integer into a string : `cat +`

Use `cat` with the integer as argument, or add the integer to an empty string
Input :

```
""+123
```

Or :

```
cat(123)
```

Output :

```
"123"
```

### 2.4.11 Transform a string into a number : `expr`

Use `expr`, the parser with a string representing a number.

- For integers, enter the string representing the integer without leading 0 for basis 10, with prefix `0x` for basis 16, `0` for basis 8 or `0b` for basis 2. Input :

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

  Output :

  ```
  123
  ```

  Input :

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

  Output :

  ```
  83
  ```

  Because : $1*8^2+2*8+3=83$ Input :

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

  Output :

```
303
```

Because : $1 * 16^2 + 2 * 16 + 15 = 303$

- For decimal numbers, use a string with a `.` or `e` inside.
  Input :

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

  Output :

  ```
  123.4567
  ```

  Input :

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

  Output :

  ```
  0.00123
  ```

- Note that `expr` more generally transforms a string into a command if the command exists.
  Input :

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

  Output :

  ```
  1
  ```

  Then, input :

  ```
  a
  ```

  Output :

  ```
  1
  ```

## 2.5 Write an integer in a $b$ basis: `convert`

`convert` or `convertir` can do different kind of conversions depending on the option given as the second argument.

To convert an integer `n` into the list of its coefficients in a basis `b`, the option is `base`. The arguments of `convert` or `convertir` are an integer `n`, `base` and `b` the value of the basis.
`convert` or `convertir` returns the list of coefficients in a `b` basis of the integer `n` Input :

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

Output :

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

To check the answer, input `expr("0173")` or `horner(revlist([3,7,1]),8)` or `convert([3,7,1],base,8)` and output is `123`
Input :

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

Output :

```
[10,11]
```

To convert the list of coefficients in a basis `b` of an integer `n`, the option is also `base`. `convert` or `convertir` returns the integer `n`.
Input :

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

Or

```
horner(revlist([3,7,1]),8)
```

Output :

```
123
```

Input :

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

Or

```
horner(revlist([10,11]),12)
```

Output :

```
142
```

## 2.6 Integers (and Gaussian Integers)

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

### 2.6.1 The factorial : `factorial`

`Xcas` can manage integers with unlimited precision, such as the following:
Input :

```
factorial(100)
```

Output :

```
9332621544394415268169923885626670049071596826438162
1468592963895217599993229915608941463976156518286253
697920827223758251185210916864000000000000000000000000
```

### 2.6.2 GCD : `gcd igcd`

`gcd` or `igcd` denotes the gcd (greatest common divisor) of several integers (for polynomials see also 2.25.7).
`gcd` or `igcd` returns the `GCD` of all integers.
Input :

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

Output :

```
3
```

Input :

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

Output :

```
3
```

Input :

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

Output :

```
3
```

We can also put as parameters two lists of same size (or a matrix with 2 lines), in this case `gcd` returns the greatest common divisor of the elements with same index (or of the same column). Input :

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

or :

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

Output :

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

**An example**
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]
```

So we have now to prove that :
if $n! = 5 + k * 7$ (for $k \in \mathbb{Z}$), $4n + 1$ and $5n + 3$ are mutually prime,
and
if $n = 5 + k * 7$ (for $k \in \mathbb{Z}$), the greatest common divisor of $4n + 1$ and $5n + 3$ is 7.

### 2.6.3 GCD : `Gcd`

`Gcd` is the inert form of `gcd`. See the section **??** for polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$ for using this instruction.
Input :

```
Gcd(18,15)
```

Output :

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

### 2.6.4 GCD of a list of integers : `lgcd`

`lgcd` has a list of integers (or of a list of polynomials) as argument.
`lgcd` returns the `gcd` of all integers of the list (or the `gcd` of all polynomials of the list).
Input :

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

Output :

```
3
```

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

### 2.6.5 The least common multiple : `lcm`

`lcm` returns the least common multiple of two integers (or of two polynomials see also 2.25.10).
Input :

```
lcm(18,15)
```

Output :

```
90
```

### 2.6.6 Decomposition into prime factors : `ifactor`

`ifactor` has an integer as parameter.
`ifactor` decomposes an integer into its prime factors.
Input :

```
ifactor(90)
```

Output :

```
2*3^2*5
```

Input :

```
ifactor(-90)
```

Output :

```
(-1)*2*3^2*5
```

### 2.6.7 List of prime factors : `ifactors`

`ifactors` has an integer (or a list of integers) as parameter.
`ifactors` decomposes the integer (or the integers of the list) into prime factors, but the result is given as a list (or a list of lists) in which each prime factor is followed by its multiplicity.
Input :

```
ifactors(90)
```

Output :

```
[2,1,3,2,5,1]
```

Input :

```
ifactors(-90)
```

Output :

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

Input :

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

Output :

```
[[2,2,3,2],[2,2,13,1]]
```

### 2.6.8 Matrix of factors : `maple_ifactors`

`maple_ifactors` has an integer $n$ (or a list of integers) as parameter.
`maple_ifactors` decomposes the integer (or the integers of the list) into prime factors, but the output follows the Maple syntax :
it is a list with +1 or -1 (for the sign) and a matrix with 2 columns and where the lines are the prime factors and their multiplicity (or a list of lists...).
Input :

```
maple_ifactors(90)
```

Output :

```
[1,[[2,1],[3,2],[5,1]]]
```

Input :

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

Output :

```
[[1,[[2,2],[3,2]]],[1,[[2,2],[13,1]]]]
```

### 2.6.9 The divisors of a number : `idivis divisors`

`idivis` or `divisors` gives the list of the divisors of a number (or of a list of numbers).
Input :

```
idivis(36)
```

Output :

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

Input :

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

Output :

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

### 2.6.10 The integer Euclidean quotient : `iquo intDiv`

`iquo` (or `intDiv`) returns the integer quotient $q$ of the Euclidean division of two integers $a$ and $b$ given as arguments. ($a = b * q + r$ with $0 \leq r < b$).
For Gaussian integers, we choose $q$ so that $b * q$ is as near by $a$ as possible and it can be proved that $r$ may be choosen so that $|r|^2 \leq |b|^2/2$.
Input :

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

Output :

`iquo` works with integers or with Gaussian integers.
Input :

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

Output :

```
3176375
```

Input :

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

Output :

```
3-2*i
```

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

### 2.6.11 The integer Euclidean remainder : `irem remain smod mods mod %`

`irem` (or `remain`) returns the integer remainder $r$ from the Euclidean division of two integers $a$ and $b$ given as arguments ($a = b * q + r$ avec $0 \leq r < b$).
For Gaussian integers, we choose $q$ so that $b * q$ is as near by $a$ as possible and it can be proved that $r$ may be choosen so that $|r|^2 \leq |b|^2/2$.
Input :

```
irem(148,5)
```

Output :

```
3
```

`irem` works with long integers or with Gaussian integers.
Example :

```
irem(factorial(148),factorial(45)+2 )
```

Output :

```
111615339728229933018338917803008301992120942047239639312
```

Another example

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

Output :

```
-4+i
```

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

`smod` or `mods` is a prefixed function and has two integers $a$ and $b$ as arguments. `smod` or `mods` returns the symetric remainder $s$ of the Euclidean division of the arguments $a$ and $b$ ($a = b * q + s$ with $-b/2 < s \leq b/2$).
Input :

```
smod(148,5)
```

Output :

```
-2
```

`mod` (or `%`) is an infixed function and has two integers $a$ and $b$ as arguments. `mod` (or `%`) returns $r\%b$ of $Z/bZ$ where $r$ is the remainder of the Euclidean division of the arguments $a$ and $b$.
Input :

```
148 mod 5
```

or

```
148 % 5
```

Output :

```
3 % 5
```

Note that the answer `3 % 5` is not an integer (3) but an element of $Z/5Z$ (see 2.31 to have the possible operations in $Z/5Z$).

### 2.6.12 Euclidean quotient and euclidean remainder of two integers : `iquorem`

`iquorem` returns the list of the quotient $q$ and the remainder $r$ of the Euclidean division between two integers $a$ and $b$ given as arguments ($a = b * q + r$ with $0 \leq r < b$).
Input :

```
iquorem(148,5)
```

Output :

```
[29,3]
```

### 2.6.13 Test of evenness : `even`

`even` takes as argument an integer `n`.
`even` returns `1` if `n` is even and returns `0` if `n` is odd.
Input :

```
even(148)
```

Output :

```
1
```

Input :

```
even(149)
```

Output :

```
0
```

### 2.6.14 Test of oddness : `odd`

`odd` takes as argument an integer `n`.
`odd` returns `1` if `n` is odd and returns `0` if `n` is even.
Input :

```
odd(148)
```

Output :

```
0
```

Input :

```
odd(149)
```

Output :

```
1
```

### 2.6.15 Test of pseudo-primality : `is_pseudoprime`

If `is_pseudoprime(n)` returns `2` (true), then `n` is prime.
If it returns 1, then `n` is pseudo-prime (most probably prime).
If it returns 0, then `n` is not prime.
DEFINITION: For numbers less than $10^{14}$, pseudo-prime and prime are equivalent. But for numbers greater than $10^{14}$, 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`)).
Input :

```
is_pseudoprime(100003)
```

Output :

```
2
```

Input :

```
is_pseudoprime(9856989898997)
```

Output :

```
2
```

Input :

```
is_pseudoprime(14)
```

Output :

```
0
```

Input :

```
is_pseudoprime(9856989898997789789)
```

Output :

```
1
```

### 2.6.16 Test of primality : `is_prime isprime isPrime`

`is_prime(n)` returns `1` (true) if `n` is prime and `0` (false) if `n` is not prime.
`isprime` returns `true` or `false`.
Use the command `pari("isprime",n,1)` to have a primality certificat (see the documentation PARI/GP with the menu `Aide->Manuels->PARI-GP`) and `pari("isprime",n,2)` to use the APRCL test.

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
```

Input :

```
pari("isprime",9856989898997789789,1)
```

This returns the coefficients giving the proof of primality by the $p-1$ Selfridge-Pocklington-Lehmer test :

```
[[2,2,1],[19,2,1],[941,2,1],[1873,2,1],[94907,2,1]]
```

Input :

```
isprime(9856989898997789789)
```

Output :

```
true
```

### 2.6.17 The smallest pseudo-prime greater than `n` : `nextprime`

`nextprime(n)` returns the smallest pseudo-prime (or prime) greater than `n`.
Input :

```
nextprime(75)
```

Output :

```
79
```

### 2.6.18 The greatest pseudo-prime less than `n` : `prevprime`

`prevprime(n)` returns the greatest pseudo-prime (or prime) less than `n`.
Input :

```
prevprime(75)
```

Output :

```
73
```

### 2.6.19 The `n`-th prime number : `ithprime`

`ithprime(n)` returns the `n`-th prime number less than 10000 (current limitation).
Input :

```
ithprime(75)
```

Output :

```
379
```

Input :

```
ithprime(1229)
```

Output :

```
9973
```

Input :

```
ithprime(1230)
```

Output :

```
ithprime(1230)
```

because `ithprime(1230)` is greater than 10000.

### 2.6.20 Bézout's Identity : `iegcd igcdex`

`iegcd(a,b)` or `igcdex(a,b)` returns the coefficients of the Bézout's Identity for two integers given as argument.
`iegcd(a,b)` or `igcdex(a,b)` returns `[u,v,d]` such that `au+bv=d` and `d=gcd(a,b)`.
Input :

```
iegcd(48,30)
```

Output :

```
[2,-3,6]
```

In other words :

$$2 \cdot 48 + (-3) \cdot 30 = 6$$

### 2.6.21 Solving au+bv=c in $\mathbb{Z}$: `iabcuv`

`iabcuv(a,b,c)` returns `[u,v]` so that `au+bv=c`.
`c` must be a multiple of `gcd(a,b)` for the existence of a solution.
Input :

```
iabcuv(48,30,18)
```

Output :

```
[6,-9]
```

### 2.6.22 Chinese remainders : `ichinrem, ichrem`

`ichinrem([a,p],[b,q])` or `ichrem([a,p],[b,q])` returns a list `[c,lcm(p,q)]` of 2 integers.
The first number `c` is such that

$$\forall k \in \mathbb{Z}, \quad d = c + k \times \mathrm{lcm}(p, q)$$

has the properties

$$d = a \pmod p, \quad d = b \pmod q$$

If `p` and `q` are coprime, a solution `d` always exists and all the solutions are congruent modulo `p*q`.
**Examples** :
Solve :

$$\begin{cases} x &= 3 \pmod 5 \\ x &= 9 \pmod{13} \end{cases}$$

Input :

```
ichinrem([3,5],[9,13])
```

or input :

```
ichrem([3,5],[9,13])
```

Output :

```
[-17,65]
```

so `x=-17 (mod 65)`
we can also input :

```
ichrem(3%5,9%13)
```

Output :

```
-17%65
```

Solve :

$$\begin{cases} x &= 3 \pmod 5 \\ x &= 4 \pmod 7 \\ x &= 1 \pmod 9 \end{cases}$$

First input :

```
tmp:=ichinrem([3,5],[4,7])
```

or input :

```
tmp:=ichrem([3,5],[4,7])
```

output :

```
[-17,35]
```

then input :

```
ichinrem([1,9],tmp)
```

or input :

```
ichrem([1,9],tmp)
```

Output :

```
[-17,315]
```

hence `x=-17 (mod 315)`
Alternative :

```
ichinrem([3%5,4%7,1%9])
```

Output :

```
-17%315
```

**Remark**
`ichrem` (or`ichinrem`)may be used to find coefficients of polynomial which class are known modulo several integers, for example find $ax+b$ modulo $315 = 5\times 7\times 9$ under the assumptions:

$$\begin{cases} a = & 3 \pmod 5 \\ a = & 4 \pmod 7 \\ a = & 1 \pmod 9 \end{cases}, \quad \begin{cases} b = & 1 \pmod 5 \\ b = & 2 \pmod 7 \\ b = & 3 \pmod 9 \end{cases}$$

Input :

```
ichrem((3x+1)%5,(4x+2)%7,(x+3)%9)
```

Output :

```
(-17%315× x+156%315
```

hence `a=-17 (mod 315)` and `b=156 (mod 315)`.

### 2.6.23 Chinese remainders for lists of integers : `chrem`

`chrem` takes as argument 2 lists of integers of the same size.
`chrem` returns a list of 2 integers.
For example, `chrem([a,b,c],[p,q,r])` returns the list `[x,lcm(p,q,r)]` where `x=a mod p` and `x=b mod q` and `x=c mod r`.
A solution `x` always exists if `p, q, r` are mutually primes, and all the solutions are equal modulo `p*q*r`.
BE CAREFULL with the order of the parameters, indeed :
`chrem([a,b],[p,q])=ichrem([a,p],[b,q])=`
`ichinrem([a,p],[b,q])`
**Examples** :
Solve :

$$\begin{cases} x = & 3 \pmod 5 \\ x = & 9 \pmod{13} \end{cases}$$

Input :

```
chrem([3,9],[5,13])
```

Output :

```
[-17,65]
```

so, `x=-17 (mod 65)`
Solve :

$$\begin{cases} x = & 3 \pmod 5 \\ x = & 4 \pmod 6 \\ x = & 1 \pmod 9 \end{cases}$$

Input :

```
chrem([3,4,1],[5,6,9])
```

Output :

```
[28,90]
```

so `x=28 (mod 90)`
**Remark**
`chrem` may be used to find coefficients of polynomial which class are known modulo several integers, for example find $ax + b$ modulo $315 = 5 \times 7 \times 9$ under the assumptions:

$$\begin{cases} a = & 3 \pmod 5 \\ a = & 4 \pmod 7 \\ a = & 1 \pmod 9 \end{cases}, \quad \begin{cases} b = & 1 \pmod 5 \\ b = & 2 \pmod 7 \\ b = & 3 \pmod 9 \end{cases}$$

Input :

```
chrem([3x+1,4x+2,x+3],[5,7,9])
```

Output :

```
[-17x+156),315]
```

hence, `a=-17 (mod 315)` et que `b=156 (mod 315)`.

### 2.6.24 Solving $a^2 + b^2 = p$ in $\mathbb{Z}$ : `pa2b2`

`pa2b2` decompose a prime integer $p$ congruent to 1 modulo 4, as a sum of squares : $p = a^2 + b^2$. The result is the list `[a,b]`.
Input :

```
pa2b2(17)
```

Output :

```
[4,1]
```

indeed $17 = 4^2 + 1^2$

### 2.6.25 The Euler indicatrix : `euler phi`

`euler` (or `phi`) returns the Euler indicatrix for a integer.
`euler(n)` (or `phi(n)`) is equal to the number of integers less than `n` and prime with `n`.
Input :

```
euler(21)
```

Output :

```
12
```

In other words E={2,4,5,7,8,10,11,13,15,16,17,19} is the set of integers less than 21 and coprime with 21. There are 12 members in this set, hence Cardinal(E)=12.

Euler has introduced this function to generalize the little Fermat theorem:

If $a$ and $n$ are mutually prime then $a^{euler(n)} = 1 \mod n$

### 2.6.26 Legendre symbol : `legendre_symbol`

If $n$ is prime, we define the Legendre symbol of $a$ written $\left(\frac{a}{n}\right)$ by :

$$\left(\frac{a}{n}\right) = \begin{cases} 0 & \text{if } a = 0 \mod n \\ 1 & \text{if } a \neq 0 \mod n \text{ and if } a = b^2 \mod n \\ -1 & \text{if } a \neq 0 \mod n \text{ and if } a \neq b^2 \mod n \end{cases}$$

Some properties

- If $n$ is prime :

$$a^{\frac{n-1}{2}} = \left(\frac{a}{n}\right) \mod n$$

- $$\begin{aligned} \left(\frac{p}{q}\right) \cdot \left(\frac{q}{p}\right) &= (-1)^{\frac{p-1}{2}}.(-1)^{\frac{q-1}{2}} \text{ if } p \text{ and } q \text{ are odd and positive} \\ \left(\frac{2}{p}\right) &= (-1)^{\frac{p^2-1}{8}} \\ \left(\frac{-1}{p}\right) &= (-1)^{\frac{p-1}{2}} \end{aligned}$$

`legendre_symbol` takes two arguments $a$ and $n$ and returns the Legendre symbol $\left(\frac{a}{n}\right)$.
Input :

```
legendre_symbol(26,17)
```

Output :

```
1
```

Input :

```
legendre_symbol(27,17)
```

Output :

```
-1
```

Input :

```
legendre_symbol(34,17)
```

Output :

```
0
```

### 2.6.27 Jacobi symbol : `jacobi_symbol`

If $n$ is not prime, the Jacobi symbol of $a$, denoted as $\left(\frac{a}{n}\right)$, is defined from the Legendre symbol and from the decomposition of $n$ into prime factors. Let

$$n = p_1^{\alpha_1}..p_k^{\alpha_k}$$

where $p_j$ is prime and $\alpha_j$ is an integer for $j = 1..k$. The Jacobi symbol of $a$ is defined by :

$$\left(\frac{a}{n}\right) = \left(\frac{a}{p_1}\right)^{\alpha_1} ... \left(\frac{a}{p_k}\right)^{\alpha_k}$$

`jacobi_symbol` takes two arguments $a$ and $n$, and it returns the Jacobi symbol $\left(\frac{a}{n}\right)$.
Input :

```
jacobi_symbol(25,12)
```

Output :

```
1
```

Input :

```
jacobi_symbol(35,12)
```

Output :

```
-1
```

Input :

```
jacobi_symbol(33,12)
```

Output :

```
0
```

## 2.7 Combinatory analysis

### 2.7.1 Factorial : `factorial` `!`

`factorial` (prefix) or `!` (postfix) takes as argument an integer $n$.
`factorial(n)` or `n!` returns $n!$.
Input :

```
factorial(10)
```

or

```
10!
```

Output :

```
3628800
```

### 2.7.2 Binomial coefficients : `binomial` `comb` `nCr`

`comb` or `nCr` or `binomial` takes as argument two integers `n` and `p`.
`comb(n,p)` or `nCr(n,p)` or `binomial(n,p)` returns $\binom{n}{p} = C_n^p$.
Input :

```
comb(5,2)
```

Output :

```
10
```

**Remark**
`binomial` (unlike `comb, nCr`) may have a third real argument, in this case `binomial(n,p,a)` returns $\binom{n}{p} a^p(1-a)^{n-p}$.

### 2.7.3 Arrangements : `perm nPr`

`perm` ou `nPr` takes as argument two integers $n$ and $p$.
`perm(n,p)` or `nPr(n,p)` returns $A_n^p$.
Input :

```
perm(5,2)
```

Output :

```
20
```

### 2.7.4 Random integers : `rand`

`rand` takes as argument an integer $n$ or no argument.

- `rand(n)` returns a random integer $p$ such that $0 \leq p < n$.
  Input :

  ```
  rand(10)
  ```

  Output for example :

  ```
  8
  ```

- `rand()` returns a random integer $p$ such that $0 \leq p < 2^{31}$ (or on 64 bits architecture $0 \leq p < 2^{63}$).
  Input :

  ```
  rand()
  ```

  Output for example :

  ```
  846930886
  ```

## 2.8 Rationals

### 2.8.1 Transform a floating point number into a rational : `exact float2rational`

`float2rational` or `exact` takes as argument a floating point number `d` and returns a rational number `q` approaching `d` such that `abs(d-q)<epsilon`. `epsilon` is defined in the `cas` configuration (`Cfg` menu) or with the `cas_setup` command.
Input :

```
float2rational(0.3670520231)
```

Output when `epsilon=1e-10`:

```
127/346
```

Input :

```
evalf(363/28)
```

Output :

```
12.9642857143
```

Input :

```
float2rational(12.9642857143)
```

Output :

```
363/28
```

If two representations are mixed, for example :

```
1/2+0.7
```

the rational is converted to a float, output :

```
1.2
```

Input :

```
1/2+float2rational(0.7)
```

Output :

```
6/5
```

### 2.8.2 Integer and fractional part : `propfrac propFrac`

`propfrac(A/B)` or `propFrac(A/B)` returns

$$q + \frac{r}{b} \text{ with } 0 \leq r < b$$

if $\frac{A}{B} = \frac{a}{b}$ with $\gcd(a,b) = 1$ and $a = bq + r$.
For rational fractions, cf. 2.28.8.
Input :

```
propfrac(42/15)
```

Output :

```
2+4/5
```

Input :

```
propfrac(43/12)
```

Output :

```
3+7/12
```

### 2.8.3 Numerator of a fraction after simplification : `numer getNum`

`numer` or `getNum` takes as argument a fraction and returns the numerator of this fraction after simplification (for rational fractions, see 2.28.2).
Input :

```
numer(42/12)
```

Or :

```
getNum(42/12)
```

Output :

```
7
```

To avoid simplifications, the argument must be quoted (for rational fractions see 2.28.1).
Input :

```
numer('42/12')
```

Or :

```
getNum('42/12')
```

Output :

```
42
```

### 2.8.4 Denominator of a fraction after simplification : `denom getDenom`

`denom` or `getDenom` takes as argument a fraction and returns the denominator of this fraction after simplification (for rational fractions see 2.28.4).
Input :

```
denom(42/12)
```

Or :

```
getDenom(42/12)
```

Output :

```
2
```

To avoid simplifications, the argument must be quoted (for rational fractions see 2.28.3).
Input :

```
denom('42/12')
```

Or :

```
getDenom('42/12')
```

Output :

```
12
```

### 2.8.5 Numerator and denominator of a fraction : `f2nd fxnd`

`f2nd` (or `fxnd`) takes as argument a fraction and returns, the list of the numerator and denominator of this fraction after simplification (for rational fractions see 2.28.5).
Input :

```
f2nd(42/12)
```

Output :

```
[7,2]
```

### 2.8.6 Simplification of a pair of integers : `simp2`

`simp2` takes as argument two integers or a list of two integers which represent a fraction (for two polynomals see 2.28.6).
`simp2` returns the list of the numerator and the denominator of an irreducible representant of this fraction (i.e. after simplification).
Input :

```
simp2(18,15)
```

Output :

```
[6,5]
```

Input :

```
simp2([42,12])
```

Output :

```
[7,2]
```

### 2.8.7 Continued fraction representation of a real : `dfc`

`dfc` takes as argument a real or a rational or a floating point number `a` and an integer `n` (or a real `epsilon`).
`dfc` returns the list of the continued fraction representation of `a` of order `n` (or with precision `epsilon` i.e. the continued fraction representation which approachs `a` or `evalf(a)` with precision `epsilon`, by default `epsilon` is the value of the `epsilon` defined in the `cas` configuration with the menu `Cfg`►`Cas Configuration`).
`convert` with the option `confrac` has a similar functionnality: in that case the value of `epsilon` is the value of the `epsilon` defined in the `cas` configuration with the menu `Cfg`►`Cas Configuration` (see 2.21.23) and the answer may be stored in an optionnal third argument.

**Remarks**

- If the last element of the result is a list, the representation is ultimaltely 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.

- if the last element of the result is not an integer, it represents a remainder $r$ ($a = a0 + 1/.... + 1/an + 1/r$). Be aware that this remainder has lost most of it's accuracy.

If `dfc(a)=[a0,a1,a2,[b0,b1]` that means :

$$a = a0 + \cfrac{1}{a1 + \frac{1}{a2 + \frac{1}{b0 + \frac{1}{b1 + \frac{1}{b0 + ...}}}}}$$

If `dfc(a)=[a0,a1,a2,r]` that means :

$$a = a0 + \cfrac{1}{a1 + \frac{1}{a2 + \frac{1}{r}}}$$

Input :

```
dfc(sqrt(2),5)
```

Output :

```
[1,2,[2]]
```

Input :

```
dfc(evalf(sqrt(2)),1e-9)
```

Or :

```
dfc(sqrt(2),1e-9)
```

Output :

```
[1,2,2,2,2,2,2,2,2,2,2,2,2]
```

Input :

```
convert(sqrt(2),confrac,'dev'
```

Output (if in the `cas` configuration `epsilon=1e-9`) :

```
[1,2,2,2,2,2,2,2,2,2,2,2,2]
```

and `[1,2,2,2,2,2,2,2,2,2,2,2,2]` is stored in `dev`.
Input :

```
dfc(9976/6961,5)
```

Output :

```
[1,2,3,4,5,43/7]
```

Input to verify:
`1+1/(2+1/(3+1/(4+1/(5+7/43))))`
Output :
`9976/6961`
Input :

```
convert(9976/6961,confrac,'l')
```

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 `l`
Input :

```
dfc(pi,5)
```

Output :

```
[3,7,15,1,292,(-113*pi+355)/(33102*pi-103993)]
```

Input :

```
dfc(evalf(pi),5)
```

Output (if floats are hardware floats, e.g. for Digits=12) :

```
[3,7,15,1,292,1.57581843574]
```

Input :

```
dfc(evalf(pi),1e-9)
```

Or :

```
dfc(pi,1e-9)
```

Or (if in the `cas` configuration `epsilon=1e-9`) :

```
convert(pi,confrac,'ll')
```

Output :

```
[3,7,15,1,292]
```

### 2.8.8 Transform a continued fraction representation into a real : `dfc2f`

`dfc2f` takes as argument a list, a continued fraction representation

- 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.
- or a list with a remainder $r$ as last element ($a = a0 + 1/.... + 1/an + 1/r$).

`dfc2f` returns the rational number or the quadratic number with the argument as continued fraction representation.
Input :

```
dfc2f([1,2,[2]])
```

Output :

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

After simplification with `normal` :

```
sqrt(2)
```

Input :

```
dfc2f([1,2,3])
```

Output :

```
10/7
```

Input :

```
normal(dfc2f([3,3,6,[3,6]]))
```

Output :

```
sqrt(11)
```

Input :

```
dfc2f([1,2,3,4,5,6,7])
```

Output :

```
9976/6961
```

Input to verify :
`1+1/(2+1/(3+1/(4+1/(5+1/(6+1/7)))))`
Output :
`9976/6961`
Input :

```
dfc2f([1,2,3,4,5,43/7])
```

Output :

```
9976/6961
```

Input to verify :
`1+1/(2+1/(3+1/(4+1/(5+7/43))))`
Output :
`9976/6961`

### 2.8.9 The $n$-th Bernoulli number : `bernoulli`

`bernoulli` takes as argument an integer $n$.
`bernoulli` returns the $n$-th Bernoulli number $B(n)$.
The Bernoulli numbers are defined by :

$$\frac{t}{e^t-1}=\sum_{n=0}^{+\infty}\frac{B(n)}{n!}t^n$$

Bernoulli polynomials $B_k$ are defined by :

$$B_0=1, \quad {B_k}'(x)=kB_{k-1}(x), \quad \int_0^1 B_k(x)dx=0$$

and the relation $B(n)=B_n(0)$ holds.
Input :

```
bernoulli(6)
```

Output :

```
1/42
```

### 2.8.10 Access to PARI/GP commands: `pari`

- `pari` with a string as first argument (the PARI command name) execute the corresponding PARI command with the remaining arguments. For example `pari("weber",1+i)` executes the PARI command `weber` with the argument `1+i`.
- `pari` without argument exports all PARI/GP functions
  - with the same command name if they are not already defined inside `Xcas`
  - with their original command name with the prefix `pari_`

  For example, after calling `pari()`, `pari_weber(1+i)` or `weber(1+i)` will execute the PARI command `weber` with the argument `1+i`.

The documentation of PARI/GP is available with the menu Help->Manuals.

## 2.9 Real numbers

### 2.9.1 Eval a real at a given precision : `evalf` and `Digits`, `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 it's 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. 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 the log of 10 in base 2.

- An expression is coerced into a floating number with the `evalf` command. `evalf` may have an optional second argument which will be used to evaluate with a given precision.
- 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.

Input :

```
1+1/2
```

Output :

```
3/2
```

Input :

```
1.0+1/2
```

Output :

```
1.5
```

Input:

```
exp(pi*sqrt(20))
```

Output :

```
exp(pi*2*sqrt(5))
```

With `evalf`, input :

```
evalf(exp(pi*2*sqrt(5)))
```

Output :

```
1263794.75367
```

Input :

```
1.1^20
```

Output :

```
6.72749994933
```

Input :

```
sqrt(2)^21
```

Output :

```
sqrt(2)*2^10
```

Input for a result with 30 digits :

```
Digits:=30
```

Input for the numeric value of $e^{\pi\sqrt{163}}$:

```
evalf(exp(pi*sqrt(163)))
```

Output :

```
0.262537412640768743999999999985e18
```

Note that `Digits` is now set to 30. If you don't want to change the value of `Digits` you may input

```
evalf(exp(pi*sqrt(163)),30)
```

### 2.9.2 Usual infixed functions on reals : `+,-,*,/,^`

`+,-,*,/,^` are the usual operators to do additions, substractions, multiplications, divisions and for raising to a power.
Input :

```
3+2
```

Output :

```
5
```

Input :

```
3-2
```

Output :

```
1
```

Input :

```
3*2
```

Output :

```
6
```

Input :

```
3/2
```

Output :

```
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 may use the square key or the cube key if your keyboard has one, for example : $3^2$ returns 9.

**Remark on non integral powers**

- If $x$ is not an integer, then $a^x = \exp(x \ln(a))$, hence $a^x$ is well-defined only for $a > 0$ if $x$ is not rational. If $x$ is rational and $a < 0$, the principal determination of the logarithm is used, leading to a complex number.
- Hence be aware of the difference between $\sqrt[n]{a}$ and $a^{\frac{1}{n}}$ when $n$ is an odd integer.
  For 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.

### 2.9.3 Usual prefixed functions on reals : `rdiv`

`rdiv` is the prefixed form of the division function.
Input :

```
rdiv(3,2)
```

Output :

```
3/2
```

Input :

```
rdiv(3.2,2.1
```

Output :

```
1.52380952381
```

### 2.9.4 $n$-th root : `root`

`root` takes two arguments : an integer $n$ and a number $a$.
`root` returns the $n$-th root of $a$ (i.e. $a^{1/n}$). If $a < 0$, the $n$-th root is a complex number of argument $2\pi/n$.
Input :

```
root(3,2)
```

Output :

```
2^(1/3)
```

Input :

```
root(3,2.0)
```

Output :

```
1.259921049892
```

Input :

```
root(3,sqrt(2))
```

Output :

```
2^(1/6)
```

### 2.9.5 Error function : `erf`

`erf` takes as argument a number $a$.
`erf` returns the floating point value of the error function at $x = a$, where the error function is defined by :

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

The normalization is choosen so that:

$$\text{erf}(+\infty) = 1, \quad \text{erf}(-\infty) = -1$$

since :

$$\int_0^{+\infty} e^{-t^2} dt = \frac{\sqrt{\pi}}{2}$$

Input :

```
erf(1)
```

Output :

```
0.84270079295
```

Input :

```
erf(1/(sqrt(2)))*1/2+0.5
```

Output :

```
0.841344746069
```

**Remark**
The relation between `erf` and `normal_cdf` is :

$$\texttt{normal\_cdf}(x) = \frac{1}{2} + \frac{1}{2}\texttt{erf}(\frac{x}{\sqrt{2}})$$

Indeed, making the change of variable $t = u * \sqrt{2}$ in

$$\text{normal\_cdf}(x) = \frac{1}{2} + \frac{1}{\sqrt{2\pi}} \int_0^x e^{-t^2/2} dt$$

gives :

$$\text{normal\_cdf}(x) = \frac{1}{2} + \frac{1}{\sqrt{\pi}} \int_0^{\frac{x}{\sqrt{2}}} e^{-u^2} du = \frac{1}{2} + \frac{1}{2}\text{erf}(\frac{x}{\sqrt{2}})$$

Check :
`normal_cdf(1)=0.841344746069`

### 2.9.6 Complementary error function: `erfc`

`erfc` takes as argument a number $a$.
`erfc` returns the value of the complementary error function at $x = a$, this function is defined by :

$$\text{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^{+\infty} e^{-t^2} dt = 1 - erf(x)$$

Hence erfc$(0) = 1$, since :

$$\int_0^{+\infty} e^{-t^2} dt = \frac{\sqrt{\pi}}{2}$$

Input :

```
erfc(1)
```

Output :

```
0.15729920705
```

Input :

```
1- erfc(1/(sqrt(2)))*1/2
```

Output :

```
0.841344746069
```

**Remark**
The relation between `erfc` and `normal_cdf` is :

$$\texttt{normal\_cdf}(x) = 1 - \frac{1}{2}\texttt{erfc}(\frac{x}{\sqrt{2}})$$

Verification :
`normal_cdf(1)=0.841344746069`

### 2.9.7 The $\Gamma$ function : `Gamma`

`Gamma` takes as argument a number $a$.
`Gamma` returns the value of the $\Gamma$ function in $a$, defined by :

$$\Gamma(x) = \int_0^{+\infty} e^{-t} t^{x-1} dt, \text{ if } x > 0$$

If $x$ is a positive integer, $\Gamma$ is computed by applying the recurrence :

$$\Gamma(x+1) = x * \Gamma(x), \quad \Gamma(1) = 1$$

Hence :

$$\Gamma(n+1) = n!$$

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)`

### 2.9.8 The $\beta$ function : `Beta`

`Beta` takes as argument two reals $a, b$.
`Beta` returns the value of the $\beta$ function at $a, b \in \mathbb{R}$, defined by :

$$\beta(x,y) = \int_0^1 t^{x-1}(1-t)^{y-1} = \frac{\Gamma(x) * \Gamma(y)}{\Gamma(x+y)}$$

Remarkable values :

$$\beta(1,1) = 1, \quad \beta(n,1) = \frac{1}{n}, \quad \beta(n,2) = \frac{1}{n(n+1)}$$

`Beta(x,y)` is defined for $x$ and $y$ positive reals (to insure the convergence of the integral) and by prolongation for $x$ and $y$ if they are not negative integers.
Input :

```
Beta(5,2)
```

Output :

```
1/30
```

Input :

```
Beta(x,y)
```

Output :

```
Gamma(x)*Gamma(y)/Gamma(x+y)
```

Input :

```
Beta(5.1,2.2)
```

Output :

```
0.0242053671402
```

### 2.9.9 Derivatives of the DiGamma fonction : `Psi`

`Psi` takes as arguments a real $a$ and an integer $n$ (by default $n = 0$).
`Psi` returns the value of the $n$-th derivative of the DiGamma function at $x = a$, where the DiGamma fonction is the first derivative of $\ln(\Gamma(x))$. This function is used to evaluated sums of rational functions having poles at integers.
Input :

```
Psi(3,1)
```

Output :

```
pi^2/6-5/4
```

If `n=0`, you may use `Psi(a)` instead of `Psi(a,0)` to compute the value of the DiGamma fonction at $x = a$.
Input :

```
Psi(3)
```

Output :

```
Psi(1)+3/2
```

Input :

```
evalf(Psi(3))
```

Output :

```
.922784335098
```

### 2.9.10 The $\zeta$ function : `Zeta`

`Zeta` takes as argument a real $x$.
`Zeta` returns for $x > 1$ :

$$\zeta(x) = \sum_{n=1}^{+\infty} \frac{1}{n^x}$$

and for $x < 1$ it's meromorphic continuation.
Input :

```
Zeta(2)
```

Output :

```
pi^2/6
```

Input :

```
Zeta(4)
```

Output :

```
pi^4/90
```

### 2.9.11 Airy functions : `Airy_Ai` and `Airy_Bi`

`Airy_Ai` and `Airy_Bi` takes as argument a real $x$.
`Airy_Ai` and `Airy_Bi` are two independant solutions of the equation

$$y'' - x * y = 0$$

They are defined by :

$$\begin{aligned}
\text{Airy\_Ai}(x) &= (1/\pi)\int_0^{\infty} \cos(t^3/3 + x * t)dt \\
\text{Airy\_Bi}(x) &= (1/\pi)\int_0^{\infty} (e^{-t^3/3} + \sin(t^3/3 + x * t))dt
\end{aligned}$$

Properties :

$$\begin{aligned}
\text{Airy\_Ai(x)} &= \text{Airy\_Ai}(0) * f(x) + \text{Airy\_Ai}'(0) * g(x) \\
\text{Airy\_Bi(x)} &= \sqrt{3}(\text{Airy\_Ai}(0) * f(x) - \text{Airy\_Ai}'(0) * g(x))
\end{aligned}$$

where $f$ and $g$ are two entire series solutions of

$$w'' - x * w = 0$$

more precisely :

$$\begin{aligned} f(x) &= \sum_{k=0}^{\infty} 3^k \left( \frac{\Gamma(k+\frac{1}{3})}{\Gamma(\frac{1}{3})} \right) \frac{x^{3k}}{(3k)!} \\ g(x) &= \sum_{k=0}^{\infty} 3^k \left( \frac{\Gamma(k+\frac{2}{3})}{\Gamma(\frac{2}{3})} \right) \frac{x^{3k+1}}{(3k+1)!} \end{aligned}$$

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
```

## 2.10 Permutations

A permutation $p$ of size $n$ is a bijection from $[0..n-1]$ on $[0..n-1]$ and is represented by the list : $[p(0), p(1), p(2)...p(n-1)]$.
For example, the permutation $p$ represented by $[1, 3, 2, 0]$ is the application from $[0, 1, 2, 3]$ on $[0, 1, 2, 3]$ defined by :

$$p(0) = 1,\ p(1) = 3,\ p(2) = 2,\ p(3) = 0$$

A cycle $c$ of size $p$ is represented by the list $[a_0, ..., a_{p-1}]$ $(0 \leq a_k \leq n-1)$ it is the permutation such that

$$c(a_i) = a_{i+1} \text{ for } (i = 0..p-2), \quad c(a_{p-1}) = a_0, \quad c(k) = k \text{ otherwise}$$

A cycle $c$ is represented by a list and a cycle decomposition is represented by a list of lists.
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 $[0, 3, 1, 2]$).

### 2.10.1 Random permutation : `randperm`

`randperm` takes as argument an integer $n$.
`randperm` returns a random permutation of $[0..n-1]$.
Input :

```
randperm(3)
```

Output :

```
[2,0,1]
```

### 2.10.2 Decomposition as a product of disjoint cycles : `permu2cycles`

`permu2cycles` takes as argument a permutation.
`permu2cycles` returns its decomposition as a product of disjoint cycles.
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]]
```

### 2.10.3 Product of disjoint cycles to permutation: `cycles2permu`

`cycles2permu` takes as argument a list of cycles.
`cycles2permu` returns the permutation (of size $n$ choosen as small as possible) that is the product of the given cycles (it is the inverse of `permu2cycles`).
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]
```

### 2.10.4 Transform a cycle into permutation : `cycle2perm`

`cycle2perm` takes on cycle as argument.
`cycle2perm` returns the permutation of size $n$ corresponding to the cycle given as argument, where $n$ is choosen as small as possible (see also `permu2cycles` and `cycles2permu`).
Input :

```
cycle2perm([1,3,5])
```

Output :

```
[0,3,2,5,4,1]
```

### 2.10.5 Transform a permutation into a matrix : `permu2mat`

`permu2mat` takes as argument a permutation $p$ of size $n$.
`permu2mat` returns the matrix of the permutation, that is the matrix obtained by permuting the rows of the identity matrix of size $n$ with the permutation $p$.
Input :

```
permu2mat([2,0,1])
```

Output :

```
[[0,0,1],[1,0,0],[0,1,0]]
```

### 2.10.6 Checking for a permutation : `is_permu`

`is_permu` is a boolean function.
`is_permu` takes as argument a list.
`is_permu` returns 1 if the argument is a permutation and returns 0 if the argument is not a permutation.
Input :

```
is_permu([2,1,3])
```

Output :

```
0
```

Input :

```
is_permu([2,1,3,0])
```

Output :

```
1
```

### 2.10.7 Checking for a cycle : `is_cycle`

`is_cycle` is a boolean function.
`is_cycle` takes a list as argument.
`is_cycle` returns 1 if the argument is a cycle and returns 0 if the argument is not a cycle.
Input :

```
is_cycle([2,1,3])
```

Output :

```
1
```

Input :

```
is_cycle([2,1,3,2])
```

Output :

```
0
```

### 2.10.8 Product of two permutations : `p1op2`

`p1op2` takes as arguments two permutations.
`p1op2` returns the permutation obtained by composition :

$$1^{\text{st}}\text{arg} \circ 2^{\text{nd}}\text{arg}$$

Input :

```
p1op2([3,4,5,2,0,1],[2,0,1,4,3,5])
```

Output :

```
[5,3,4,0,2,1]
```

**Warning**
Composition is done using the standard mathematical notation, that is the permutation given as second argument is performed first.

### 2.10.9 Composition of a cycle and a permutation : `c1op2`

`c1op2` takes as arguments a cycle and a permutation.
`c1op2` returns the permutation obtained by composition :

$$1^{\text{st}}\text{arg} \circ 2^{\text{nd}}\text{arg}$$

Input :

```
c1op2([3,4,5],[2,0,1,4,3,5])
```

Output :

```
[2,0,1,5,4,3]
```

**Warning**
Composition is done using the standard mathematical notation, that is the permutation given as second argument is performed first.

### 2.10.10 Composition of a permutation and a cycle : `p1oc2`

`p1oc2` takes as arguments a permutation and a cycle.
`p1oc2` returns the permutation obtained by composition :

$$1^{\text{st}}\text{arg} \circ 2^{\text{nd}}\text{arg}$$

Input :

```
p1oc2([3,4,5,2,0,1],[2,0,1])
```

Output :

```
[4,5,3,2,0,1]
```

**Warning**
Composition is done using the standard mathematical notation, that is the cycle given as second argument is performed first.

### 2.10.11 Product of two cycles : `c1oc2`

`c1oc2` takes as arguments two cycles.
`c1oc2` returns the permutation obtained by composition :

$$1^{\text{st}}\text{arg} \circ 2^{\text{nd}}\text{arg}$$

Input :

```
c1oc2([3,4,5],[2,0,1])
```

Output :

```
[1,2,0,4,5,3]
```

**Warning**
Composition is done using the standard mathematical notation, that is the cycle given as second argument is performed first.

### 2.10.12 Signature of a permutation : `signature`

`signature` takes as argument a permutation.
`signature` returns the signature of the permutation given as argument.
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 : $(-1)^{k+1}$.
Input :

```
signature([3,4,5,2,0,1])
```

Output :

```
-1
```

Indeed `permu2cycles([3,4,5,2,0,1])=[[0,3,2,5,1,4]]`.

### 2.10.13 Inverse of a permutation : `perminv`

`perminv` takes as argument a permutation.
`perminv` returns the permutation that is the inverse of the permutation given as argument.
Input :

```
perminv([1,2,0])
```

Output

```
[2,0,1]
```

### 2.10.14 Inverse of a cycle : `cycleinv`

`cycleinv` takes as argument a cycle.
`cycleinv` returns the cycle that is the inverse of the cycle given as argument.
Input :

```
cycleinv([2,0,1])
```

Output

```
[1,0,2]
```

### 2.10.15 Order of a permutation : `permuorder`

`permuorder` takes as argument a permutation.
`permuorder` returns the order $k$ of the permutation $p$ given as argument, that is the smallest integer $m$ such that $p^m$ is the identity.
Input :

```
permuorder([0,2,1])
```

Output

```
2
```

Input :

```
permuorder([3,2,1,4,0])
```

Output

```
6
```

### 2.10.16 Group generated by two permutations : `groupermu`

`groupermu` takes as argument two permutations `a` and `b`.
`groupermu` returns the group of the permutations generated by `a` and `b`.
Input :

```
groupermu([0,2,1,3],[3,1,2,0])
```

Output

```
[[0,2,1,3],[3,1,2,0],[0,1,2,3],[3,2,1,0]]
```

## 2.11 Complex numbers

Note that complex numbers are also used to represent a point in the plan or a 1-d function graph.

### 2.11.1 Usual complex functions : `+,-,*,/,^`

`+,-,*,/,^` are the usual operators to perform additions, substractions, multiplications, divisions and for raising to an integer or a fractional power.
Input :

```
(1+2*i)^2
```

Output :

```
-3+4*i
```

### 2.11.2 Real part of a complex number : `re real`

`re` (or `real`) takes as argument a complex number (resp a point $A$).
`re` (or `real`) returns the real part of this complex number (resp the projection on the $x$ axis of $A$).
Input :

```
re(3+4*i)
```

Output :

```
3
```

### 2.11.3 Imaginary part of a complex number : `im imag`

`im` (or `imag`) takes as argument a complex number (resp a point $A$).
`im` (or `imag`) returns imaginary part of this complex number (resp the projection on the $y$ axis of $A$).
Input :

```
im(3+4*i)
```

Output :

```
4
```

### 2.11.4 Write a complex as `re(z)+i*im(z)` : `evalc`

`evalc` takes as argument a complex number `z`.
`evalc` returns of this complex number, written as `re(z)+i*im(z)`.
Input :

```
evalc(sqrt(2)*exp(i*pi/4))
```

Output :

```
1+i
```

### 2.11.5 Modulus of a complex number : `abs`

`abs` takes as argument a complex number.
`abs` returns the modulus of this complex number.
Input :

```
abs(3+4*i)
```

Output :

```
5
```

### 2.11.6 Argument of a complex number : `arg`

`arg` takes as argument a complex number.
`arg` returns the argument of this complex number.
Input :

```
arg(3+4.i)
```

Output :

```
atan(4/3)
```

### 2.11.7 The normalized complex number : `normalize unitV`

`normalize` or `unitV` takes as argument a complex number.
`normalize` or `unitV` returns the complex number divided by the modulus of this complex number.
Input :

```
normalize(3+4*i)
```

Output :

```
(3+4*i)/5
```

### 2.11.8 Conjuguate of a complex number : `conj`

`conj` takes as argument a complex number.
`conj` returns the complex conjuguate of this complex number.
Input :

```
conj(3+4*i)
```

Output :

```
3-4*i
```

### 2.11.9 Multiplication by the complex conjugate : `mult_c_conjugate`

`mult_c_conjugate` takes as argument an complex expression.
If this expression has a complex denominator, `mult_c_conjugate` multiplies the numerator and the denominator of this expression by the complex conjuguate of the denominator.
If this expression has not a complex denominator, `mult_c_conjugate` multiplies the numerator and the denominator of this expression by the complex conjuguate of the numerator.
Input :

```
mult_c_conjugate((2+i)/(2+3*i))
```

Output :

```
(2+i)*(2+3*(-i))/((2+3*(i))*(2+3*(-i)))
```

Input :

```
mult_c_conjugate((2+i)/2)
```

Output :

```
(2+i)*(2+-i)/(2*(2+-i))
```

### 2.11.10 Barycenter of complex numbers : `barycentre`

**See also : ??** and **??**.
`barycentre` takes as argument two lists of the same size (resp. a matrix with two columns):

- the elements of the first list (resp column) are points $A_j$ or complex numbers $a_j$ (the affixes of the points),
- the elements of the second list (resp column) are real coefficients $\alpha_j$ such that $\sum \alpha_j \neq 0$.

`barycentre` returns the barycenter point of the points $A_j$ weighted by the real coefficients $\alpha_j$. If $\sum \alpha_j = 0$, `barycentre` returns an error.
**Warning** To have a complex number in the output, the input must be :
`affixe(barycentre(...,...))` because `barycentre(...,...)` returns a point, not a complex number.
Input :

```
affixe(barycentre([1+i,1-i],[1,1]))
```

Or :

```
affixe(barycentre([[1+i,1],[1-i,1]]))
```

Output :

```
i
```

## 2.12 Algebraic expressions

### 2.12.1 Evaluate an expression : `eval`

`eval` is used to evaluate an expression. Since `Xcas` always evaluate expressions entered in the commandline, `eval` is mainly used to evaluate a sub-expression in the equation writer.
Input :

```
a:=2
```

Output :

```
2
```

Input :

```
eval(2+3*a)
```

or

```
2+3*a
```

Output :

```
8
```

### 2.12.2 Evaluate 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`.

### 2.12.3 Prevent evaluation : `quote hold '`

A quoted subexpression (either with `'` or with the `quote` or `hold`) command will not be evaluated.
**Remark** `a:=quote(a)` (or `a:=hold(a)`) is equivalent to `purge(a)` (for the sake of Maple compatibility). It returns the value of this variable (or the hypothesis done on this variable).
Input :

```
a:=2;quote(2+3*a)
```

or

```
a:=2;'2+3*a'
```

Output :

```
(2,2+3.a)
```

### 2.12.4 Force evaluation : `unquote`

`unquote` is used to evaluate inside a quoted expression.
For example in an affectation, the variable is automatically quoted (not evaluated) so that the user does not have to quote it explicitely each time he want to modify it's value. In some circumstances, you might however want to evaluate it.
Input:

```
purge(b);a:=b;unquote(a):=3
```

Output :

```
b contains 3, hence a evals to 3
```

### 2.12.5 Distributivity : `expand fdistrib`

`expand` or `fdistrib` takes as argument an expression.
`expand` ou `fdistrib` returns the expression where the distributivity of the multiplication with respect to the addition is applied.
Input :

```
expand((x+1)*(x-2))
```

Or :

```
fdistrib((x+1)*(x-2))
```

Output :

```
x^2-2*x+x-2
```

### 2.12.6 Canonical form : `canonical_form`

`canonical_form` takes as argument a trinomial of second degree.
`canonical_form` returns the canonical form of the argument.
Example :
Find the canonical form of :

$$x^2 - 6x + 1$$

Input :

```
canonical_form(x^2-6*x+1)
```

Outpout :

```
(x-3)^2-8
```

### 2.12.7 Multiplication by the conjugate quantity : `mult_conjugate`

`mult_conjugate` takes as argument an expression with a denominator or a numerator supposed to contain a square root :

- if the denominator contains a square root,
  `mult_conjugate` multiplies the numerator and the denominator of the expression by the conjugate quantity of the denominator.
- otherwise, if the numerator contains a square root,
  `mult_conjugate` multiplies the numerator and the denominator of this expression by the conjugate quantity of the numerator.

Input :

```
mult_conjugate((2+sqrt(2))/(2+sqrt(3)))
```

Output :

```
(2+sqrt(2))*(2-sqrt(3))/((2+sqrt(3))*(2-sqrt(3)))
```

Input :

```
mult_conjugate((2+sqrt(2))/(sqrt(2)+sqrt(3)))
```

Output :

```
(2+sqrt(2))*(-sqrt(2)+sqrt(3))/
((sqrt(2)+sqrt(3))*(-sqrt(2)+sqrt(3)))
```

Input :

```
mult_conjugate((2+sqrt(2))/2)
```

Output :

```
(2+sqrt(2))*(2-sqrt(2))/(2*(2-sqrt(2)))
```

### 2.12.8 Separation of variables : `split`

`split` takes two arguments : an expression depending on two variables and the list of these two variables.
If the expression may be factorized into two factors where each factor depends only of one variable, `split` returns the list of this two factors, otherwise it returns the list `[0]`.
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]
```

### 2.12.9 Factorisation : `factor`

`factor` takes as argument an expression.
`factor` factorizes this expression on the field of it's coefficients, with the addition of $i$ in complex mode. If `sqrt` is enabled in the Cas configuration, polynomials of order 2 are factorized in complex mode or in real mode if the discriminant is positive.
**Examples**

1. Factorize $x^4 - 1$ over $\mathbb{Q}$.
   Input :

   ```
   factor(x^4-1)
   ```

   Output :

   ```
   (x^2+1)*(x+1)*(x-1)
   ```

   The coefficients are rationals, hence the factors are polynomials with rationals coefficients.

2. Factorize $x^4 - 1$ over $\mathbb{Q}[i]$
   To have a complex factorisation, check `complex` in the `cas` configuration (red button displaying the status line).
   Input :

```
factor(x^4-1)
```

Output :

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

3. Factorize $x^4 + 1$ over $\mathbb{Q}$
   Input :

   ```
   factor(x^4+1)
   ```

   Output :

   ```
   x^4+1
   ```

   Indeed $x^4 + 1$ has no factor with rational coefficients.

4. Factorize $x^4 + 1$ over $\mathbb{Q}[i]$
   Check `complex` in the `cas` configuration (red button rouge displaying the status line).
   Input :

   ```
   factor(x^4-1)
   ```

   Output :

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

5. Factorize $x^4 + 1$ 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 and input :

   ```
   solve(x^4+1,x)
   ```

   Output :

   ```
   [sqrt(2)/2+(i)*sqrt(2)/2,sqrt(2)/2+(i)*(-(sqrt(2)/2)),
   -sqrt(2)/2+(i)*sqrt(2)/2,-sqrt(2)/2+(i)*(-(sqrt(2)/2))]
   ```

   The roots depends on $\sqrt{2}$. Uncheck complex mode in the Cas configuration and input :

   ```
   factor(x^4+1,sqrt(2))
   ```

   Output :

   ```
   (x^2+sqrt(2)*x+1)*(x^2+(-(sqrt(2)))*x+1)
   ```

   To factorize over $\mathbb{C}$, check `complex` in the `cas` configuration and input `cFactor(x^4+1,sqrt(2))` (cf `cFactor`).

### 2.12.10 Complex factorisation : `cFactor`

`cFactor` takes as argument an expression.
`cFactor` factorizes this expression on the field $\mathbb{Q}[i] \subset \mathbb{C}$ (or over the complexified field of the coefficients of the argument) even if you are in real mode.
**Examples**

1. Factorize $x^4 - 1$ over $\mathbb{Z}[i]$.
   Input :
   ```
   cFactor(x^4-1)
   ```
   Output :
   ```
   -((x+-i)*((-i)*x+1)*((-i)*x+i)*(x+1))
   ```
2. Factorize $x^4 + 1$ over $\mathbb{Z}[i]$.
   Input :
   ```
   cFactor(x^4+1)
   ```
   Output :
   ```
   (x^2+i)*(x^2+-i)
   ```
3. For a complete factorization of $x^4 + 1$, check the sqrt box in the Cas configuration or input :
   ```
   cFactor(x^4+1,sqrt(2))
   ```
   Output :
   ```
   sqrt(2)*1/2*(sqrt(2)*x+1-i)*(sqrt(2)*x-1+i)*sqrt(2)*
         1/2*(sqrt(2)*x+1+i)*(sqrt(2)*x-1-i)
   ```

### 2.12.11 Zeros of an expression : `zeros`

`zeros` takes as argument an expression depending on $x$.
`zeros` returns a list of values of $x$ where the expression vanishes. The list may be incomplete in exact mode if the expression is not polynomial or if intermediate factorizations have irreducible factors of order strictly greater than 2.
In real mode, (complex box unchecked in the Cas configuration or `complex_mode:=0`), only reals zeros are returned. In (`complex_mode:=1`) reals and complex zeros are returned. See also `cZeros` to get complex zeros in real mode.
Input in real mode :
```
zeros(x^2+4)
```
Output :

```
[]
```

Input in complex mode :

```
zeros(x^2+4)
```

Output :

```
[-2*i,2*i]
```

Input in real mode :

```
zeros(ln(x)^2-2)
```

Output :

```
[exp(sqrt(2)),exp(-(sqrt(2)))]
```

Input in real mode :

```
zeros(ln(y)^2-2,y)
```

Output :

```
[exp(sqrt(2)),exp(-(sqrt(2)))]
```

Input in real mode :

```
zeros(x*(exp(x))^2-2*x-2*(exp(x))^2+4)
```

Output :

```
[[log(sqrt(2)),2]
```

### 2.12.12 Complex zeros of an expression : `cZeros`

`cZeros` takes as argument an expression depending on $x$.
`cZeros` returns a list of comnplex values of $x$ where the expression vanishes. The list may be incomplete in exact mode if the expression is not polynomial or if intermediate factorizations have irreducible factors of order strictly greater than 2.
Input in real or complex mode :

```
cZeros(x^2+4)
```

Output :

```
[-2*i,2*i]
```

Input :

```
cZeros(ln(x)^2-2)
```

Output :

```
[exp(sqrt(2)),exp(-(sqrt(2)))]
```

Input :

```
cZeros(ln(y)^2-2,y)
```

Output :

```
[exp(sqrt(2)),exp(-(sqrt(2)))]
```

Input :

```
cZeros(x*(exp(x))^2-2*x-2*(exp(x))^2+4)
```

Output :

```
[[log(sqrt(2)),log(-sqrt(2)),2]
```

### 2.12.13 Normal form : `normal`

`normal` takes as argument an expression. The expression is considered as a rational fraction with respect to generalized identifiers (either true identifiers or transcendental functions replaced by a temporary identifiers) with coefficients in $\mathbb{Q}$ or $\mathbb{Q}[i]$ or in an algebraic extension (e.g. $\mathbb{Q}[\sqrt{2}]$). `normal` returns the expanded irreducible representation of this rational fraction. See also `ratnormal` for pure rational fractions or `simplify` if the transcendental functions are not algebraically independant.
Input :

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

Output :

```
x^2-1
```

**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.

### 2.12.14 Simplify : `simplify`

`simplify` simplifies an expression. It behaves like `normal` for rational fractions and algebraic extensions. For expressions containing transcendental functions, `simplify` tries first to rewrite them in terms of algebraically independant transcendental functions. For trigonometric expressions, this requires radian mode (check `radian` in the `cas` configuration or input `angle_radian:=1`).
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 :

```
4*(cos(x))^2-2
```

### 2.12.15 Normal form for rational fractions : `ratnormal`

`ratnormal` rewrites an expression using it's irreductible representation. The expression is viewed as a multivariate rational fraction with coefficients in $\mathbb{Q}$ (or $\mathbb{Q}[i]$). The variables are generalized identifiers which are assumed to be algebraically independant. 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 dependant transcendental functions.
Input :

```
ratnormal((x^3-1)/(x^2-1))
```

Output :

```
(x^2+x+1)/(x+1)
```

Input :

```
ratnormal((-2x^3+3x^2+5x-6)/(x^2-2x+1))
```

Output :

```
(-2*x^2+x+6)/(x-1)
```

### 2.12.16 Substitue a variable by a value : `subst`

`subst` takes two or three arguments :

- an expression depending on a variable, an equality (variable=value of substitution) or a list of equalities.
- an expression depending on a variable, a variable or a list of variables, a value or a list of values for substitution.

`subst` returns the expression with the substitution done. Note that `subst` does not quote it's argument, hence in a normal evaluation process, the substitution variable should be purged otherwise it will be replaced by it's assigned value before substitution is done.
Input :

```
subst(a^2+1,a=2)
```

or :

```
subst(a^2+1,a,2)
```

Output (if the variable `a` is purged else first input `purge(a)`) :

```
5
```

Input :

```
subst(a^2+b,[a,b],[2,1])
```

Or :

```
subst(a^2+b,[a=2,b=1])
```

Output (if the variables `a` and `b` are purged else first input `purge(a,b)`) :

```
2^2+1
```

`subst` may also be used to make a change of variable in an integral. In this case the `integrate` command should be quoted (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 argument of `Int` or `integrate` even you are integrating with respect to `x`.
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

```
integrate(sin(t)*sqrt(t)*1/2*1/t*sqrt(t),t,0,(pi/2)^2)
```

Input :

```
subst('integrate(sin(x^2)*x,x)',x=sqrt(t))
```

Or :

```
subst(Int(sin(x^2)*x,x),x=sqrt(t))
```

Output

```
integrate(sin(t)*sqrt(t)*1/2*1/t*sqrt(t),t)
```

### 2.12.17 Substitue a variable by a value (Maple and Mupad compatibility) : `subs`

In `Maple` and in `Mupad`, one would use the `subs` command to substitue a variable by a value in an expression. But the order of the arguments differ between `Maple` and `Mupad`. Therefore, to achieve compatibility, `Xcas` `subs` command arguments order depends on the mode

- In `Maple` mode, `subs` takes two arguments : an equality (variable=substitution value) and the expression.
  To substitue several variables in an expression, use a list of equality (variable names = substitution value) as first argument.
- In `Mupad` or `Xcas` or `TI`, `subs` takes two or three arguments : an expression and an equality (variable=substitution value) or an expression, a variable name and the substitution value.
  To substitue several variables, `subs` takes two ou three arguments :
  - an expression of variables and a list of (variable names = substitution value),
  - an expression of variables, a list of variables and a list of their substitution values.

`subs` returns the expression with the substitution done. Note that `subs` does not quote it's argument, hence in a normal evaluation process, the substitution variable should be purged otherwise it will be replaced by it's assigned value before substitution is done.
Input in `Maple` mode (if the variable `a` is purged else input `purge(a)`) :

```
subs(a=2,a^2+1)
```

Output

```
2^2+1
```

Input in `Maple` mode (if the variables `a` and `b` are purged else input `purge(a,b)`):

```
subs([a=2,b=1],a^2+b)
```

Output :

```
2^2+1
```

Input :

```
subs(a^2+1,a=2)
```

or :

```
subs(a^2+1,a,2)
```

Output (if the variable `a` is purged else input `purge(a)`) :

```
5
```

Input :

```
subs(a^2+b,[a=2,b=1])
```

or :

```
subs(a^2+b,[a,b],[2,1])
```

Output (if the variables `a` and `b` are purged else input `purge(a,b)`) :

```
2^2+1
```

### 2.12.18 Evaluate a primitive at boundaries: `preval`

`preval` takes three arguments : an expression `F` depending on the variable `x`, and two expressions `a` and `b`.
`preval` computes $F_{|x=b} - F_{|x=a}$.
`preval` is used to compute a definite integral when the primitive $F$ of the integrand *f* is known. Assume for example that `F:=int(f,x)`, then `preval(F,a,b)` is equivalent to `int(f,x,a,b)` but does not require to compute again `F` from `f` if you change the values of $a$ or $b$.
Input :

```
preval(x^2+x,2,3)
```

Output :

```
6
```

### 2.12.19 Sub-expression of an expression : `part`

`part` takes two arguments : an expression and an integer $n$.
`part` evaluate the expression and then returns the $n$-th sub-expression of this expression.
Input :

```
part(x^2+x+1,2)
```

Output :

```
x
```

Input :

```
part(x^2+(x+1)*(y-2)+2,2)
```

Output :

```
(x+1)*(y-2)
```

Input :

```
part((x+1)*(y-2)/2,2)
```

Output :

```
y-2
```

## 2.13 Values of $u_n$

### 2.13.1 Array of values of a sequence : `tablefunc`

`tablefunc` is a command that should be used inside a spreadsheet (opened with `Alt+t`), it returns a template to fill two columns, with the table of values of a function. If the step value is 1, `tablefunc(ex,n,n0,1)`, where `ex` is an expression depending on `n`, will fill the spreadsheet with the values of the sequence $u_n = ex$ for $n = n0,\ n0+1,\ n0+2, .....$

**Example** : display the values of the sequence $u_n = \sin(n)$
Select a cell of a spreadsheet (for example `C0`) and input in the command line :

```
tablefunc(sin(n),n,0,1)
```

Output :

```
two columns :  n and sin(n)
```

- in the column C: the variable name `n`, the value of the step (this value should be equal to 1 for a sequence), the value of `n0` (here 0), then a recurrence formula (`C2+C$1`, ...).
- in the column D: `sin(n)`, `"Tablefunc"`, then a recurrence formula.
- For each row, the values of the sequence $\mathtt{u_n} = \sin(\mathtt{n})$ correspond to the values of `n` starting from `n=n0` (here 0).

### 2.13.2 Table of values and graph of a recurrent sequence : `tableseq` and `plotseq`

`tableseq` is a command that should be used inside a spreadsheet (opened with `Alt+t`), it returns a template to fill one column with $\mathtt{u_0},\ \mathtt{u_{n+1}} = \mathtt{f}(\mathtt{u_n})$ (one-term recurrence) or more generally $u_0, ..., u_k,\ \ u_{n+k+1} = f(u_n, u_{n+1}, ..., u_{n+k})$. The template fills the column starting from the selected cell, or starting from 0 if the whole column was selected.
See also `plotseq` (section 3.13) for a graphic representation of a one-term recurrence sequence.

**Examples** :

- display the values of the sequence $u_0 = 3.5,\ u_n = \sin(u_{n-1})$
  Select a cell of the spreadsheet (for example `B0`) and input in the command line :

  ```
  tableseq(sin(n),n,3.5)
  ```

  Output :

  ```
  a column with sin(n), n, 3.5 and the formula
             evalf(subst(B$0,B$1,B2))
  ```

  You get the values of the sequence $\mathtt{u_0} = 3.5,\ \mathtt{u_n} = \mathtt{sin}(\mathtt{u_{n-1}})$ in the column `B`.

- display the values of the Fibonacci sequence $u_0 = 1, u_1 = 1$ $u_{n+2} = u_n + u_{n+1}$
  Select a cell, say `B0`, and input in the command line

  ```
  tableseq(x+y,[x,y],[1,1])
  ```

  This fills the B column sheet with

  | row | B |
  |---|---|
  | 0 | x+y |
  | 1 | x |
  | 2 | y |
  | 3 | 1 |
  | 4 | 1 |
  | 5 | 2 |
  | .. | .. |
  | 7 | 5 |
  | .. | .. |

## 2.14 Operators or infixed functions

An operator is an infixed function.

### 2.14.1 Usual operators :`+, -, *, /, ^`

`+, -, *, /, ^` are the operators to do additions, substractions, multiplications, divisions and for raising to a power.

### 2.14.2 `Xcas` operators

- `$` is the infixed version of `seq` for example :
  `(2^k)$(k=0..3)= seq(2^k,k=0..3)=(1,2,4,8)` (do not forget to put parenthesis around the arguments),
- `mod` or `%` to define a modular number,
- `@` to compose functions for example : `(f@g)(x)=f(g(x))`,
- `@@` to compose a function many times (like a power, replacing multiplication by composition), for example : `(f@@3)(x)=f(f(f(x)))`,
- `minus union intersect` to have the difference, the union and the intersection of two sets,
- `->` to define a function,
- `:=` `=>` to store an expression in a une variable (it is the infixed version of `sto` and the argument order is permuted for `:=`), for example : `a:=2` or `2=>a` or `sto(2,a)`.

- =< 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.

### 2.14.3 Define an operator: `user_operator`

`user_operator` takes as argument :

- a string : the name of the operator,
- a function of two variables with values in $\mathbb{R}$ or in `true, false`,
- an option `Binary` for the definition or `Delete` to annulate this definition.

`user_operator` returns 1 if the definition is done and else returns 0.

**Example 1**
Let $R$ be defined on $\mathbb{R}$ by $x\ R\ y = x * y + x + y$.
To define the law $R$, input :

```
user_operator("R",(x,y)->x*y+x+y,Binary)
```

Output :

```
1
```

Input :

```
5 R 7
```

Do not forget to put spaces around `R`.
Output :

```
47
```

**Example 2**
Let $S$ by defined on $\mathbb{N}$ by :
for $x$ and $y$ integers, $x\ S\ y$ <=> $x$ and $y$ are not coprime.
To define the law $S$, input :

```
user_operator("S",(x,y)->(gcd(x,y))!=1,Binary)
```

Output :

```
1
```

Input :

```
5 S 7
```

Do not forget to put spaces around `S`.
Output :

```
0
```

Input :

```
8 S 12
```

Do not forget to put spaces around `S`.
Output :

```
1
```

## 2.15 Functions and expressions with symbolic variables

### 2.15.1 Difference between function and expression

A function `f` is defined for example by :
`f(x):=x^2-1` or by `f:=x->x^2-1`
that is to say, for all $x$, $f(x)$ is equal to the expression $x^2 - 1$. In that case, to have the value of $f$ for $x = 2$, input :`f(2)`.
But if the input is `g:=x^2-1`, then `g` is a variable where the expression $x^2 - 1$ is stored. In that case, to have the value of $g$ for $x = 2$, input : `subst(g,x=2)` ($g$ is an expression depending on $x$).

When a command expects a function as argument, this argument should be either the definition of the function (e.g. `x->x^2-1`) or a variable name assigned to a function (e.g. `f` previously defined by e.g. `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, for example, by `g:=x^2-1`), or the evaluation of a function. e.g. `f(x)` if `f` is a previously defined function, for example, by `f(x):=x^2-1`).

### 2.15.2 Transform an expression into a fonction : `unapply`

`unapply` is used to transform an expression into a function.
`unapply` takes two arguments an expression and the name of a variable.
`unapply` returns the function defined by this expression and this variable.

**Warning** when a function is defined, the right member of the affectation is not evaluated, hence `g:=sin(x+1); f(x):=g` does not defined the function $f : x \to sin(x + 1)$ but defines the function $f : x \to g$. To defined the former function, `unapply` should be used, like in the following example:
Input :

```
g:= sin(x+1); f:=unapply(g,x)
```

Output :

```
(sin(x+1), (x)->sin(x+1))
```

hence, the variable `g` is assigned to a symbolic expression and the variable `f` is assigned to a function.
Input :

```
unapply(exp(x+2),x)
```

Output :

```
(x)->exp(x+2)
```

Input :

```
f:=unapply(lagrange([1,2,3],[4,8,12]),x)
```

Output :

```
(x)->4+4*(x-1)
```

Input :

```
f:=unapply(integrate(log(t),t,1,x),x)
```

Output :

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

Input :

```
f:=unapply(integrate(log(t),t,1,x),x)
```

```
f(x)
```

Output :

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

**Remark** Suppose that $f$ is a function of 2 variables $f : (x, w) \to f(x, w)$, and that $g$ is the function defined by $g : w \to h_w$ where $h_w$ is the function defined by $h_w(x) = f(x, w)$.
`unapply` is also used to define $g$ with `Xcas`.
Input :

```
f(x,w):=2*x+w
```

```
g(w):=unapply(f(x,w),x)
```

```
g(3)
```

Output :

```
x->2· x+3
```

### 2.15.3 Top and leaves of an expression : `sommet feuille op`

An operator is an infixed function : for exemple '+' is an operator and 'sin' is a function.
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 of the function (see also 2.37.11).
The instruction `sommet` (resp `feuille` (or `op`)) returns the top (resp. the list of the leaves) of an expression.
Input :

```
sommet(sin(x+2))
```

Output :

```
'sin'
```

Input :

```
sommet(x+2*y)
```

Output :

```
'+'
```

Input :

```
feuille(sin(x+2))
```

Or :

```
op(sin(x+2))
```

Output :

```
x+2
```

Input :

```
feuille(x+2*y)
```

Or :

```
op(x+2*y)
```

Output :

```
(x,2*y)
```

**Remark**
Suppose that a function is defined by a program, for example let us define the `pgcd` function :

```
pgcd(a,b):={local r; while (b!=0)
{r:=irem(a,b);a:=b;b:=r;} return a;}
```

Then input :

```
sommet(pgcd)
```

Output :

```
'program'
```

Then input :

```
feuille(pgcd)[0]
```

Output :

```
(a,b)
```

Then input :

```
feuille(pgcd)[1]
```

Output :

```
(0,0) or (15,25) if the last input was pgcd(15,25)
```

Then input :

```
feuille(pgcd)[2]
```

Output :

```
The body of the program :  {local r;....return(a);}
```

## 2.16 Functions

### 2.16.1 Context-dependant functions.

#### Operators + and −

+ (resp −) is an infixed function and `'+'` (resp `'-'`) is a prefixed fonction. The result depends on the nature of its arguments.
Examples with + (all examples except the last one work also with − instead of +) :

- input (1,2)+(3,4) or (1,2,3)+4 or 1+2+3+4 or '+'(1,2,3,4), output 10,
- input 1+i+2+3*i or '+'(1,i,2,3*i), output 3+4*i,
- 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 [[2,4],[6,8]],
- input [1,2,3]+4 or '+'([1,2,3],4), output poly1[1,2,7],
- input [1,2,3]+(4,1) or '+'([1,2,3],4,1), output poly1[1,2,8],
- input "Hel"+"lo" or '+'("Hel","lo"), output "Hello".

#### Operator $\star$

$\star$ is an infixed function and `'*'` is a prefixed function. The result depends on the nature of its arguments.
Examples with $\star$ :

- input (1,2)*(3,4) or (1,2,3)*4=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 (scalar product),
- input [1,2]*[3,4]='*'([1,2],[3,4]), output 11 (scalar product),
- input [[1,2],[3,4]]* [[1,2],[3,4]], output [[7,10],[15,22]],
- 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+6*i.

**Operator** `/`

`/` is an infixed function and `'/'` is a prefixed function. The result depends of the nature of its arguments.
Examples with `/` :

- input [10,2,3]/[4,1], output invalid dim
- input [1,2]/[3,4] or '/'([1,2],[3,4]), output [1/3,1/2],
- input 1/[[1,2],[3,4]] or '/'(1,[[1,2],[3,4]], output [[-2,1],[3/2,(-1)/2]],
- input [[1,2],[3,4]]*1/ [[1,2],[3,4]], output [[1,0],[0,1]],
- input [[1,2],[3,4]]/ [[1,2],[3,4]], output [[1,1],[1,1]] (division term by term),

### 2.16.2 Usual functions

- `max` takes as argument two real numbers and returns their maximum,
- `min` takes as argument two real numbers and returns their minimum,
- `abs` takes as argument a complex number and returns the modulus of the complex parameter (the absolute value if the complex is real),
- `sign` takes as argument a real number and returns its sign (+1 if it is positive, 0 if it is null, and -1 if it is negative),
- `floor` (or `iPart`) takes as argument a real number $r$, and returns the largest integer $\leq r$,
- `round` takes as argument a real number and returns it's nearest integer,
- `ceil` or `ceiling` takes as argument a real number and returns the smallest integer $\geq r$
- `frac` (or `fPart`) takes as argument a real number and returns its fractionnal part,
- `trunc` takes as argument a real number and returns the integer equal to the real without its fractionnal part,
- `id` is the identity function,
- `sq` is the square function,
- `sqrt` is the squareroot function,
- `exp` is the exponential function,
- `log` or `ln` is the natural logarithm function,
- `log10` is the base-10 logarithm function,
- `logb` is the logarithm function where the second argument is the basis of the logarithm: `logb(7,10)=log10(7)=log(7)/log(10)`,

- `sin` (resp. `cos`, `tan`) is the sinus function, cosinus function, tangent function,
- `cot, sec, csc` are the cotangeant, secant, cosecant function
- `asin` (or `arcsin`), `acos` (or `arccos`), `atan` (or `arctan`), `acot, asec, acsc` are the inverse trigonometric functions (see section 2.21.1 for more info on trigonometric functions)
- `sinh` (resp. `cosh`, `tanh`) is the hyperbolic sinus function, cosinus function, tangent function,
- `asinh` or `arcsinh` (resp `acosh` or `arccosh`, `atanh` or `arctanh`) is the inverse function of `sinh` (resp `cosh`, `tanh`)

### 2.16.3 Defining algebraic functions

#### Defining a function from $\mathbb{R}^p$ to $\mathbb{R}$

For $p = 1$, e.g. for $f : (x) \to x * \sin(x)$, input :

```
f(x):=x*sin(x)
```

Or :

```
f:=x->x*sin(x)
```

Output :

```
(x)->x*sin(x)
```

If $p > 1$, e.g. for $f : (x, y) \to x * \sin(y)$, input :

```
f(x,y):=x*sin(y)
```

Or :

```
f:=(x,y)->x*sin(y)
```

Output :

```
(x,y)->x*sin(y)
```

**Warning !!!** the expression after `->` is not evaluated. You should use `unapply` if you expect the second member to be evaluated before the function is defined.

#### Defining a function from $\mathbb{R}^p$ to $\mathbb{R}^q$

For example:

- To define the function $h : (x, y) \to (x * \cos(y), x * \sin(y))$.
  Input :

  ```
  h(x,y):=(x*cos(y),x*sin(y))
  ```

  Output :

```
expr(" (x,y)->{
x*cos(y),x*sin(y);
}",0)
```

- To 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)];}
  ```

  Output :

  ```
  (x,y)->{return([x*cos(y),x*sin(y)]);}
  ```

**Warning !!!** the expression after `->` is not evaluated.

**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 we 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 `->` (or `:=`) is not evaluated, we should not define $g(t)$ by `g(t):=y->f(t,y)`, we have to use the `unapply` command.

For example, assuming that $f : (x, y) \rightarrow x \sin(y)$ and $g(t) : y \rightarrow f(t, y)$, input :

```
f(x,y):=x*sin(y);g(t):=unapply(f(t,y),y)
```

Output :

```
((x,y)->x*sin(y), (t)->unapply(f(t,y),y))
```

Input

```
g(2)
```

Output :

```
y->2· sin(y)
```

Input

```
g(2)(1)
```

Output :

```
2· sin(1)
```

Next example, suppose that the function $h : (x, y) \rightarrow [x * \cos(y), x * \sin(y)]$ is defined, and we 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):=unapply(h(x,t),x)
```

Output :

```
(t)->unapply(h(x,t),x)
```

Input

```
k(2)
```

Output :

```
(x)->(x*cos(2),x*sin(2))
```

Input

```
k(2)(1)
```

Output :

```
(2*cos(1),2*sin(1))
```

### 2.16.4 Composition of two functions: `@`

With `Xcas`, the composition of functions is done with the infixed operator `@`.
Input :

```
(sq@sin+id)(x)
```

Output :

```
(sin(x))^2+x
```

Input :

```
(sin@sin)(pi/2)
```

Output :

```
sin(1)
```

### 2.16.5 Repeted function composition: `@@`

With `Xcas`, the repeted composition of a function with itself $n \in \mathbb{N}$ times is done with the infixed operator `@@`.
Input :

```
(sin@@3)(x)
```

Output :

```
sin(sin(sin(x)))
```

Input :

```
(sin@@2)(pi/2)
```

Output :

```
sin(1)
```

### 2.16.6 Define a fonction with the history : `as_function_of`

If an entry defined the variable `a` and if in a later entry defines the variable `b` (supposed to be dependant on `a`), then `c:=as_function_of(b,a)` will define a function `c` such that `c(a)=b`.
Input :

```
a:=sin(x)
```

Output :

```
sin(x)
```

Input :

```
b:=sqrt(1+a^2)
```

Output :

```
sqrt(1+sin(x)^2)
```

Input :

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

Output :

```
(a)->
{ local NULL;
return(sqrt(1+a^2));
}
```

Input :

```
c(x)
```

Output :

```
sqrt(1+x^2)
```

Input :

```
a:=2
```

Output :

```
2
```

Input :

```
b:=1+a^2
```

Output :

```
5
```

Input :

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

Output :

```
(a)->
{ local NULL;
return(sqrt(1+a^2));
}
```

Input :

```
c(x)
```

Output :

```
1+x^2
```

**Warning !!**
If the variable `b` has been assigned several times, the first affectation of `b` following the last affectation 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.
Input for example :
`a:=2` then
`b:=2*a+1` then
`b:=3*a+2` then
`c:=as_function_of(b,a)`
Output :

```
(a)-> {local NULL; return(2*a+1);}
```

i.e. `c(x)` is equal to `2*x+1`.
But, input :
`a:=2` then
`b:=2*a+1` then
`a:=2` then
`b:=3*a+2` then
`c:=as_function_of(b,a)`
Output :

```
(a)-> {local NULL; return(3*a+2);}
```

i.e. `c(x)` is equal to `3*x+2`.
Hence the line where `a` is define must be revalided before the good definition of `b`.

## 2.17 Derivation and applications.

### 2.17.1 Functional derivative : `function_diff`

`function_diff` takes a function as argument.
`function_diff` returns the derivative function of this function.
Input :

```
function_diff(sin)
```

Output :

```
(` x`)->cos(` x`)
```

Input :

```
function_diff(sin)(x)
```

Output :

```
cos(x)
```

Input :

```
f(x):=x^2+x*cos(x)
function_diff(f)
```

Output :

```
(` x`)->2*` x`+cos(` x`)+` x`*(-(sin(` x`)))
```

Input :

```
function_diff(f)(x)
```

Output :

```
cos(x)+x*(-(sin(x)))+2*x
```

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 in conjonction with `unapply` :

```
g:=unapply(diff(f(x),x),x)
g(x)
```

Output :

```
cos(x)+x*(-(sin(x)))+2*x
```

**Warning !!!**
In `Maple` mode, 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`).

### 2.17.2 Length of an arc : `arcLen`

`arcLen` takes four arguments : an expression $ex$ (resp a list of two expressions $[ex1, ex2]$), the name of a parameter and two values $a$ and $b$ of this parameter.
`arcLen` computes the length of the curve define by the equation $y = f(x) = ex$ (resp by $x = ex1, y = ex2$) when the parameter values varies from $a$ to $b$, using the formula `arcLen(f(x),x,a,b)=`
`integrate(sqrt(diff(f(x),x)^2+1),x,a,b)`
or
`integrate(sqrt(diff(x(t),t)^2+diff(y(t),t)^2),t,a,b).`

**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 :

  ```
  -1/4*log(sqrt(5)-2)-(-(sqrt(5)))/2
  ```

- 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 :

  ```
  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
  ```

### 2.17.3 Maximum and minimum of an expression: `fMax` `fMin`

`fMax` and `fMin` take one or two arguments : an expression of a variable and the name of this variable (by default `x`).
`fMax` returns the abscissa of a maximum of the expression.
`fMin` returns the abscissa of a minimum of the expression.
Input :

```
fMax(sin(x),x)
```

Or :

```
fMax(sin(x))
```

Or :

```
fMax(sin(y),y)
```

Output :

```
pi/2
```

Input :

```
fMin(sin(x),x)
```

Or :

```
fMin(sin(x))
```

Or :

```
fMin(sin(y),y)
```

Output :

```
-pi/2
```

Input :

```
fMin(sin(x)^2,x)
```

Output :

```
0
```

### 2.17.4 Table of values and graph : `tablefunc` and `plotfunc`

`tablefunc` is a special command that should be run from inside the spreadsheet. It returns the evaluation of an expression $ex$ depending on a variable $x$ for $x = x_0,\ x_0 + h, ....$ :

`tablefunc(ex,x,x_0,h)` or `tablefunc(ex,x)`

In the latter case, the default value for `x0` is the default minimum value of $x$ from the graphic configuration and the default value for the step $h$ is 0.1 times the difference between the default maximum and minimum values of $x$ (from the graphic configuration).
Example: type `Alt+t` to open a spreadsheet if none are open. Then select a cell of the spreadsheet (for example `C0`) and to get the table of `"sinus"`, input in the command line of the spreadsheet :

```
tablefunc(sin(x),x)
```

This will fill two columns with the numeric value of `x` and `sin(x)` :

- in the first column the variable `x`, the value of the step `h` (1.0), the minimum value of $x$ (-5.0), then a formula, for example `=C2+C$1`, and the remaining rows of the column is filled by pasting this formula.
- in the next column the function `sin(x)`, the word "Tablefunc", a formula, for example `=evalf(subst(D$0,C$0,C2))`, and the remaining rows of the column are filled by pasting this formula.

Hence the values of `sin(x)` are on the same rows as the values of `x`. Note that the step and begin value and the expression may be easily changed by modifying the correspondant cell.
The graphic representation may be plotted with the `plotfunc` command (see 3.2.1).

### 2.17.5 Derivative and partial derivative

`diff` or `derive` may have one or two arguments to compute a first order derivative (or first order partial derivative) of an expression or of a list of expressions, or several arguments to compute the $n$-th partial derivative of an expression or list of expressions.

#### Derivative and first order partial derivative : `diff derive deriver`

`diff` (or `derive`) takes two arguments : an expression and a variable (resp a vector of the variable name) (see several variable functions in 2.50). If only one argument is provided, the derivative is taken with respect to `x`
`diff` (or `derive`) returns the derivative (resp a vector of derivative) of the expression with respect to the variable (resp with respect to each variable) given as second argument.
Examples :

- Compute :
$$\frac{\partial(x.y^2.z^3+x.y.z)}{\partial z}$$
Input :
```
diff(x*y ^2*z^3+x*y*z,z)
```
Output :

```
x*y^2*3*z^2+x*y
```

- 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^2*z^3+y*z, x*2*y*z^3+x*z, x*y^2*3*z^2+x*y]
  ```

**Derivative and $n$-th order partial derivative :** `diff derive deriver`

`derive` (or `diff`) may take more than two arguments : an expression and the names of the derivation variables (each variable may be followed by $$n$ to indicate the number $n$ of derivations).
`diff` returns the partial derivative of the expression with respect to the variables given after the first argument.

The notation $ is usefull if you want to derive $k$ times with respect to the same variable, instead of entering $k$ times the same variable name, one enters the variable name followed by `$k`, for example `x$3` instead of `(x,x,x)`. Each variable may be followed by a $, for example `diff(exp(x*y),x$3,y$2,z)` is the same as `diff(exp(x*y),x,x,x,y,y,z)`
**Examples**

- Compute :
  $$\frac{\partial^2(x.y^2.z^3+x.y.z)}{\partial x\partial z}$$
  Input :

  ```
  diff(x*y ^2*z^3+x*y*z,x,z)
  ```

  Output :

  ```
  y^2*3*z^2+y
  ```

- Compute :
  $$\frac{\partial^3(x.y^2.z^3+x.y.z)}{\partial x\partial^2 z}$$
  Input :

  ```
  diff(x*y ^2*z^3+x*y*z,x,z,z)
  ```

  Or input :

  ```
  diff(x*y ^2*z^3+x*y*z,x,z$2)
  ```

  Output :

```
y^2*3*2*z
```

- 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 :

  ```
  (-24*x^3+48*x)/(x^8+8*x^6+24*x^4+32*x^2+16)
  ```

**Remark**

- Note the difference between `diff(f,x,y)` and `diff(f,[x,y])` :
  `diff`$(f, x, y)$ returns $\dfrac{\partial^2(f)}{\partial x \partial y}$ and
  `diff`$(f, [x, y])$ returns $[\dfrac{\partial(f)}{\partial x}, \dfrac{\partial(f)}{\partial y}]$
- 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: the variable name to define the $f_1$ function and the differentiation variable. The right way to define a derivative is either with `function_diff` or:

  ```
  f1:=unapply(diff(f(x),x),x)
  ```

## 2.18 Integration

### 2.18.1 Antiderivative and definite integral : `integrate int Int`

`integrate` (or `int`) compute the 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.

`integrate` (or `int` or `Int`) takes one, two or four arguments.

- with one or two arguments
  an expression or an expression and the name of a variable (by default `x`), `integrate` (or `int`) returns a primitive of the expression with respect to the variable given as second argument.
  Input :

  ```
  integrate(x^2)
  ```

Output :

```
x^3/3
```

Input :

```
integrate(t^2,t)
```

Output :

```
t^3/3
```

- with four arguments :
  an expression, a name of a variable and the bounds of the definite integral, `integrate` (or `int`) returns the exact value of the definite integral if the computation was successfull or an unevaluated integral otherwise.
  Input :

  ```
  integrate(x^2,x,1,2)
  ```

  Output :

  ```
  7/3
  ```

  Input :

  ```
  integrate(1/(sin(x)+2),x,0,2*pi)
  ```

  Output after simplification (with the `simplify` command) :

  ```
  2*pi*sqrt(3)/3
  ```

`Int` is the inert form of `integrate`, it prevents evaluation for example to avoid a symbolic computation that might not be successfull if you just want a numeric evaluation.
Input :

```
evalf(Int(exp(x^2),x,0,1))
```

Or :

```
evalf(int(exp(x^2),x,0,1))
```

Output :

```
1.46265174591
```

**Exercise 1**
Let

$$f(x) = \frac{x}{x^2-1} + \ln(\frac{x+1}{x-1})$$

Find a primitive of $f$.
Input :

```
int(x/(x^2-1)+ln((x+1)/(x-1)))
```

Output :

```
x*log((x+1)/(x-1))+log(x^2-1)+1/2*log(2*x^2/2-1)
```

Or define the function `f`, input :

```
f(x):=x/(x^2-1)+ln((x+1)/(x-1))
```

then input :

```
int(f(x))
```

Output of course the same result.
**Warning**
For `Xcas`, `log` is the natural logarithm (like `ln`), as `log10` is 10-basis logarithm
**Exercise 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*((3*x^2+2)/(-(2*(x^3+x)))+-3/2*atan(x))
```

**Exercise 3**
Compute :

$$\int \frac{1}{\sin(x)+\sin(2\cdot x)}\,dx$$

Input :

```
integrate(1/(sin(x)+sin(2*x )))
```

Output :

```
(1/-3*log((tan(x/2))^2-3)+1/12*log((tan(x/2))^2))*2
```

### 2.18.2 Discrete summation: `sum`

`sum` takes two or four arguments :

- four arguments
  an expression, the name of the variable (for example `n`), and the bounds (for example `a` and `b`).
  `sum` returns the discrete sum of this expression with respect to the variable from $a$ to $b$.
  Input :

```
sum(1,k,-2,n)
```

Output :

```
n+1+2
```

Input :

```
normal(sum(2*k-1,k,1,n))
```

Output :

```
n^2
```

Input :

```
sum(1/(n^2),n,1,10)
```

Output :

```
1968329/1270080
```

Input :

```
sum(1/(n^2),n,1,+(infinity))
```

Output :

```
pi^2/6
```

Input :

```
sum(1/(n^3-n),n,2,10)
```

Output :

```
27/110
```

Input :

```
sum(1/(n^3-n),n,1,+(infinity))
```

Output :

```
1/4
```

This result comes from the decomposition of `1/(n^3 - n)`.
Input :

```
partfrac(1/(n^3-n))
```

Output :

```
1/(2*(n+1))-1/n+1/(2*(n-1))
```

Hence :
$$\sum_{n=2}^{N} -\frac{1}{n} = -\sum_{n=1}^{N-1} \frac{1}{n+1} = -\frac{1}{2} - \sum_{n=2}^{N-2} \frac{1}{n+1} - \frac{1}{N}$$
$$\frac{1}{2} * \sum_{n=2}^{N} \frac{1}{n-1} = \frac{1}{2} * (\sum_{n=0}^{N-2} \frac{1}{n+1}) = \frac{1}{2} * (1 + \frac{1}{2} + \sum_{n=2}^{N-2} \frac{1}{n+1})$$
$$\frac{1}{2} * \sum_{n=2}^{N} \frac{1}{n+1} = \frac{1}{2} * (\sum_{n=2}^{N-2} \frac{1}{n+1} + \frac{1}{N} + \frac{1}{N+1})$$
After simplification by $\sum_{n=2}^{N-2}$, it remains :
$$-\frac{1}{2} + \frac{1}{2} * (1 + \frac{1}{2}) - \frac{1}{N} + \frac{1}{2} * (\frac{1}{N} + \frac{1}{N+1}) = \frac{1}{4} - \frac{1}{2N(N+1)}$$
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.

- two arguments
  an expression of one variable (for example $f$) and the name of this variable (for example $x$).
  `sum` returns the discrete antiderivative of this expression, i.e. an expression $G$ such that $G_{|x=n+1} - G_{|x=n} = f_{|x=n}$.
  Input :

```
sum(1/(x*(x+1)),x)
```

  Output :

```
-1/x
```

### 2.18.3 Riemann sum : `sum_riemann`

`sum_riemann` takes two arguments : an expression depending of two variables and the list of the name of these two variables.
`sum_riemann(expression(n,k),[n,k])` returns in the neighboorhoud of $n = +\infty$ an equivalent of $\sum_{k=1}^{n} expression(n,k)$ (or of $\sum_{k=0}^{n-1} expression(n,k)$ or of $\sum_{k=1}^{n-1} expression(n,k)$) when the sum is looked as a Riemann sum associated to a continue function defined on [0,1] or returns `"it is probably not a Riemann sum"` when the resarch is unavailing.
**Exercise 1**
Suppose $S_n = \sum_{k=1}^{n} \frac{k^2}{n^3}$.
Compute $\lim_{n\to+\infty} S_n$.
Input :

```
sum_riemann(k^2/n^3,[n,k])
```

Output :

```
1/3
```

**Exercise 2**
Suppose $S_n = \sum_{k=1}^{n} \frac{k^3}{n^4}$.
Compute $\lim_{n \to +\infty} S_n$.
Input :

```
sum_riemann(k^3/n^4,[n,k])
```

Output :

```
1/4
```

**Exercise 3**
Compute $\lim_{n \to +\infty} (\frac{1}{n+1} + \frac{1}{n+2} + ... + \frac{1}{n+n})$.
Input :

```
sum_riemann(1/(n+k),[n,k])
```

Output :

```
log(2)
```

**Exercise 4**
Suppose $S_n = \sum_{k=1}^{n} \frac{32n^3}{16n^4 - k^4}$.
Compute $\lim_{n \to +\infty} S_n$.
Input :

```
sum_riemann(32*n^3/(16*n^4-k^4),[n,k])
```

Output :

```
2*atan(1/2)+log(3)
```

## 2.18.4 Integration by parts : `ibpdv` et `ibpu`

`ibpdv`

`ibpdv` is used to search the primitive of an expression written as $u(x).v'(x)$.
`ibpdv` takes two arguments :

- an expression $u(x).v'(x)$ and $v(x)$ (or a list of two expressions $[F(x), u(x) * v'(x)]$ and $v(x)$),
- or an expression $g(x)$ and 0 (or a list of two expressions $[F(x), g(x)]$ and 0).

`ibpdv` returns :

- if $v(x) \neq 0$, the list $[u(x).v(x), -v(x).u'(x)]$ (or $[F(x)+u(x).v(x), -v(x).u'(x)]$),

- if the second argument is zero, a primitive of the first argument $g(x)$ (or $F(x)$+a primitive of $g(x)$) :
  hence, `ibpdv(g(x),0)` returns a primitive `G(x)` of `g(x)` or
  `ibpdv([F(x),g(x)],0)` returns `F(x)+G(x)` where `diff(G(x))=g(x)`.

Hence, `ibpdv` returns the terms computed in an integration by parts, with the possibility of doing several `ibpdv` succesively.
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.
Input :

```
ibpdv(ln(x),x)
```

Output :

```
[x.ln(x),-1]
```

then

```
ibpdv([x.ln(x),-1],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 `ibpdv` successively).
For example :
`ibpdv((log(x))^2,x) = [x*(log(x))^2,-(2*log(x))]`
it remains to integrate `-(2*log(x))`, the input :
`ibpdv(ans(),x)` or input :
`ibpdv([x*(log(x))^2,-(2*log(x))],x)`
Output :
`[x*(log(x))^2+x*(-(2*log(x))),2]`
and it remains to integrate `2`, hence input `ibpdv(ans(),0)` or
`ibpdv([x*(log(x))^2+x*(-(2*log(x))),2],0)`.
Output : `x*(log(x))^2+x*(-(2*log(x)))+2*x`

`ibpu`

`ibpu` is used to search the primitive of an expression written as $u(x).v'(x)$ `ibpu` takes two arguments :

- an expression $u(x).v'(x)$ and $u(x)$ (or a list of two expressions $[F(x), u(x)*v'(x)]$ and $u(x)$),
- an expression $g(x)$ and 0 (or a list of two expressions $[F(x)g(x)]$ and 0).

`ibpu` returns :

- if $u(x) \neq 0$, the list $[u(x).v(x), -v(x).u'(x)]$ (or returns the list $[F(x) + u(x).v(x), -v(x).u'(x)]$),
- if the second argument is zero, a primitive of the first argument $g(x)$ (or $F(x)$+a primitive of $g(x)$):
  `ibpu(g(x),0)` returns `G(x)` where `diff(G(x))=g(x)` or
  `ibpu([F(x),g(x)],0)` returns `F(x)+G(x)` where `diff(G(x))=g(x)`.

Hence, `ibpu` returns the terms computed in an integration by parts, with the possibility of doing several `ibpu` succesively.
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 `ibpu` command with the answer as first argument and a new $u(x)$ (or 0 to terminate the integration) as second argument.
Input :

```
ibpu(ln(x),ln(x))
```

Output :

```
[x.ln(x),-1]
```

then

```
ibpu([x.ln(x),-1],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 `ibpu` successively).
For example :
`ibpu((log(x))^2,log(x)) = [x*(log(x))^2,-(2*log(x))]`
it remains to integrate `-(2*log(x))`, hence input :
`ibpu(ans(),log(x))` or input :
`ibpu([x*(log(x))^2,-(2*log(x))],log(x))`
Output :
`[x*(log(x))^2+x*(-(2*log(x))),2]`
it remains to integrate `2`, hence input :
`ibpu(ans(),0)` or input :
`ibpu([x*(log(x))^2+x*(-(2*log(x))),2],0)`.
Output : `x*(log(x))^2+x*(-(2*log(x)))+2*x`

### 2.18.5 Change of variables : `subst`

See the `subst` command in the section 2.12.16.

## 2.19 Limits

### 2.19.1 Limites : `limit`

`limit` compute the limit of an expression in a finite or infinite point. It is also possible with an optional argument to compute unidirectional limit (1 for right limit and -1 for left limit .
`limit` takes three or four arguments :
an expression, the name of a variable (for example `x`), the limit point (for example `a`) and an optional argument, by default `0`, to indicate if the limit is unidirectional. This argument is equal to `-1` for a left limit (`x<a`) or is equal to `1` for a right limit (`x>a`) or is equal to `0` for a limite.
`limit` returns the limit of the expression when the variable (for example `x`) approachs the limit point (for example `a`).
**Remark**
It is also possible to put `x=a` as argument instead of `x,a`, hence : `limit` takes also as arguments an expression depending of a variable, an equality (variable =value of the limit point) and perhaps 1 or -1 to indicate the direction.
Input :

```
limit(1/x,x,0,-1)
```

or

```
limit(1/x,x=0,-1)
```

Output :

```
-(infinity)
```

Input :

```
limit(1/x,x,0,1)
```

or

```
limit(1/x,x=0,1)
```

Output :

```
+(infinity)
```

Input :

```
limit(1/x,x,0,0)
```

or

```
limit(1/x,x,0)
```

or

```
limit(1/x,x=0)
```

Output :

```
infinity
```

Hence, `abs(1/x)` approachs $+\infty$ when $x$ approachs 0.

**Exercises** :

- Find for $n > 2$, the limit when $x$ approachs 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 when $x$ approachs $+\infty$ of :

  $$\sqrt{x + \sqrt{x + \sqrt{x}}} - \sqrt{x}$$

  Input :

  ```
  limit(sqrt(x+sqrt(x+sqrt(x)))-sqrt(x),x=+infinity)
  ```

  Output :

  ```
  1/2
  ```

- Find the limit when $x$ approachs 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 :

  ```
  -1/6
  ```

**Remark**

To compute limit, it is better sometimes to quote the first argument.
Input :

```
limit('(2*x-1)*exp(1/(x-1))',x=+infinity)
```

Note that the first argument is quoted, because it is better that this argument is not simplified (i.e. not evaluated).
Output :

```
+(infinity)
```

### 2.19.2 Integral and limit

Just two examples :

- Find the limit, when $a$ approachs $+\infty$, of :

  $$\int_2^a \frac{1}{x^2}\, dx$$

  Input :

  ```
  limit(integrate(1/(x^2),x,2,a),a,+(infinity))
  ```

  Output (if `a` is not formal then input `purge(a)`) :

  ```
  1/2
  ```

- Find the limit, when $a$ approachs $+\infty$, of :

  $$\int_2^a (\frac{x}{x^2-1} + \ln(\frac{x+1}{x-1}))\, dx$$

  Input :

  ```
  limit(integrate(x/(x^2-1)+log((x+1)/(x-1)),x,2,a),
  
                      a,+(infinity))
  ```

  Output (if `a` is not formal then input `purge(a)`) :

  ```
  +(infinity)
  ```

## 2.20 Rewriting transcendental and trigonometric expressions

### 2.20.1 Expand a transcendental and trigonometric expression : `texpand tExpand`

`texpand` or `tExpand` takes as argument an expression containing transcendental or trigonometric functions.
`texpand` or `tExpand` expands these functions, like simultaneous calling `expexpand`, `lnexpand` and `trigexpand`, for example, $\ln(x^n)$ becomes $n\ln(x)$, $\exp(nx)$ becomes $\exp(x)^n$, $\sin(2x)$ becomes $2\sin(x)\cos(x)$...
**Examples** :

- 1. Expand $\cos(x+y)$.
     Input :

     ```
     texpand(cos(x+y))
     ```

Output :

```
cos(x)*cos(y)-sin(x)*sin(y)
```

2. Expand $\cos(3x)$.
   Input :

   ```
   texpand(cos(3*x))
   ```

   Output :

   ```
   4*(cos(x))^ 3-3*cos(x)
   ```

3. Expand $\dfrac{\sin(3*x)+\sin(7*x)}{\sin(5*x)}$.
   Input :

   ```
   texpand((sin(3*x)+sin(7*x))/sin(5*x))
   ```

   Output

   ```
   (4*(cos(x))^2-1)*(sin(x)/(16*(cos(x))^4-
   12*(cos(x))^2+1))/sin(x)+(64*(cos(x))^6-
   80*(cos(x))^4+24*(cos(x))^2-1)*sin(x)/
   (16*(cos(x))^4-12*(cos(x))^2+1)/sin(x)
   ```

   Output, after a simplification with `normal(ans())` :

   ```
   4*(cos(x))^2-2
   ```

- 1. Expand $\exp(x+y)$.
     Input :

     ```
     texpand(exp(x+y))
     ```

     Output :

     ```
     exp(x)*exp(y)
     ```

  2. Expand $\ln(x \times y)$.
     Input :

     ```
     texpand(log(x*y))
     ```

     Output :

     ```
     log(x)+log(y)
     ```

  3. Expand $\ln(x^n)$.
     Input :

     ```
     texpand(ln(x^n))
     ```

     Output :

     ```
     n*ln(x)
     ```

  4. Expand $\ln((e2)+\exp(2*\ln(2))+exp(\ln(3)+\ln(2)))$.
     Input :

     ```
     texpand(log(e^2)+exp(2*log(2))+exp(log(3)+log(2)))
     ```

Output :

```
6+3*2
```

Or input :

```
texpand(log(e^2)+exp(2*log(2)))+
  lncollect(exp(log(3)+log(2)))
```

Output :

```
12
```

- Expand $\exp(x + y) + \cos(x + y) + \ln(3x2)$.
  Input :

  ```
  texpand(exp(x+y)+cos(x+y)+ln(3*x^2))
  ```

  Output :

  ```
  cos(x)*cos(y)-sin(x)*sin(y)+exp(x)*exp(y)+
                  ln(3)+2*ln(x)
  ```

### 2.20.2 Combine terms of same type : `combine`

`combine` takes two arguments : an expression and the name of a function or class of functions `exp,log,ln, sin,cos,trig`.
Whenever possible, `combine` put together subexpressions corresponding to the second argument:

- `combine(expr,ln)` or `combine(expr,log)` gives the same result as `lncollect(expr)`
- `combine(expr,trig)` or `combine(expr,sin)` or `combine(expr,cos)` gives the same result as `tcollect(expr)`.

Input :

```
combine(exp(x)*exp(y)+sin(x)*cos(x)+ln(x)+ln(y),exp)
```

Output :

```
exp(x+y)+sin(x)*cos(x)+ln(x)+ln(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 :

```
exp(y)*exp(x)+(sin(2*x))/2+ln(x)+ln(y)
```

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 :

```
exp(x)*exp(y)+sin(x)*cos(x)+ln(x*y)
```

## 2.21 Trigonometry

### 2.21.1 Trigonometric functions

- `sin` is the sine function,
- `cos` is the cosine function,
- `tan` is the tangent function (`tan(x)= sin(x)/cos(x)`),
- `cot` is the cotangent function (`cot(x)= cos(x)/sin(x)`),
- `sec` is the secant function (`sec(x)= 1/cos(x)`),
- `csc` is the cosecant function (`csc(x) = 1/sin(x)`),
- `asin` or `arcsin`, `acos` or `arccos`, `atan` or `arctan`, `acot`, `asec`, `acsc` are the inverse trigonometric functions. The latter are defined by:
  1. `asec(x) = acos(1/x)`,
  2. `acsc(x) = asin(1/x)`,
  3. `acot(x) = atan(1/x)`.

### 2.21.2 Expand a trigonometric expression : `trigexpand`

`trigexpand` takes as argument an expression containing trigonometric functions.
`trigexpand` expands sums, differences and products by an integer inside the trigonometric functions
Input :

```
trigexpand(cos(x+y))
```

Output :

```
cos(x)*cos(y)-sin(x)*sin(y)
```

### 2.21.3 Linearize a trigonometric expression : `tlin`

`tlin` takes as argument an expression containing trigonometric functions.
`tlin` linearize products and integer powers of the trigonometric functions (e.g. in terms of $\sin(n.x)$ and $\cos(n.x)$)
**Examples**

- Linearize $\cos(x) * \cos(y)$.
  Input :
  ```
  tlin(cos(x)*cos(y))
  ```
  Output :
  ```
  1/2*cos(x-y)+1/2*cos(x+y)
  ```
- Linearize $\cos(x)3$.
  Input :
  ```
  tlin(cos(x)^3)
  ```
  Output :
  ```
  3/4*cos(x)+1/4*cos(3*x)
  ```
- Linearize $4\cos(x)2 - 2$.
  Input :
  ```
  tlin(4*cos(x)^2-2)
  ```
  Output :
  ```
  2*cos(2*x)
  ```

### 2.21.4 Put together sine and cosine of the same angle : `tcollect` `tCollect`

`tcollect` or `tCollect` takes as argument an expression containing trigonometric functions.
`tcollect` first linearizes this expression (e.g. in terms of $\sin(n.x)$ and $\cos(n.x)$), then, puts together sine and cosine of the same angle.
Input :
```
tcollect(sin(x)+cos(x))
```
Output :
```
sqrt(2)*cos(x-pi/4)
```
Input :
```
tcollect(2*sin(x)*cos(x)+cos(2*x))
```
Output :
```
sqrt(2)*cos(2*x-pi/4)
```

### 2.21.5 Simplify : `simplify`

`simplify` simplifies the expression.
As with all automatic simplifications, do not expect miracles, you will have to use specific rewriting rules if it does not work.
Input :

```
simplify((sin(3*x)+sin(7*x))/sin(5*x))
```

Output :

```
4*(cos(x))^2-2
```

**Warning** `simplify` is more efficient in `radian` mode (check `radian` in the `cas` configuration or input `angle_radian:=1`).

### 2.21.6 Transform arccos into arcsin : `acos2asin`

`acos2asin` takes as argument an expression containing inverse trigonometric functions.
`acos2asin` replaces $\arccos(x)$ by $\frac{\pi}{2} - \arcsin(x)$, in this expression.
Input :

```
acos2asin(acos(x)+asin(x))
```

Output after simplification :

```
pi/2
```

### 2.21.7 Transform arccos into arctan : `acos2atan`

`acos2atan` takes as argument an expression containing inverse trigonometric functions.
`acos2atan` replaces $\arccos(x)$ by $\frac{\pi}{2} - \arctan(\frac{x}{\sqrt{1-x2}})$, in this expression.
Input :

```
acos2atan(acos(x))
```

Output :

```
pi/2-atan(x/sqrt(1-x^2))
```

### 2.21.8 Transform arcsin into arccos : `asin2acos`

`asin2acos` takes as argument an expression containing inverse trigonometric functions.
`asin2acos` replaces $\arcsin(x)$ by $\frac{\pi}{2} - \arccos(x)$, in this expression.
Input :

```
asin2acos(acos(x)+asin(x))
```

Output after simplification :

```
pi/2
```

### 2.21.9 Transform arcsin into arctan : `asin2atan`

`asin2atan` takes as argument an expression containing inverse trigonometric functions.
`asin2atan` replaces $\arcsin(x)$ by $\arctan(\frac{x}{\sqrt{1-x2}})$, in this expression.
Input :

```
asin2atan(asin(x))
```

Output :

```
atan(x/sqrt(1-x^2))
```

### 2.21.10 Transform arctan into arcsin : `atan2asin`

`atan2asin` takes as argument an expression containing inverse trigonometric functions. `atan2asin` replaces $\arctan(x)$ by $\arcsin(\frac{x}{\sqrt{1+x2}})$, in this expression.
Input :

```
atan2asin(atan(x))
```

Output :

```
asin(x/sqrt(1+x^2))
```

### 2.21.11 Transform arctan into arccos : `atan2acos`

`atan2acos` takes as argument an expression containing inverse trigonometric functions.
`atan2acos` replaces $\arctan(x)$ by $\frac{\pi}{2} - \arccos(\frac{x}{\sqrt{1+x2}})$, in this expression.
Input :

```
atan2acos(atan(x))
```

Output :

```
pi/2-acos(x/sqrt(1+x^2))
```

### 2.21.12 Transform complex exponentials into sin and cos : `sincos` `exp2trig`

`sincos` or `exp2trig` takes as argument an expression containing complex exponentials.
`sincos` or `exp2trig` rewrites this expression in terms of sin and cos.
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)
```

### 2.21.13 Transform tan(x) into sin(x)/cos(x) : `tan2sincos`

`tan2sincos` takes as argument an expression containing trigonometric functions.
`tan2sincos` replaces $\tan(x)$ by $\frac{\sin(x)}{\cos(x)}$, in this expression.
Input :

```
tan2sincos(tan(2*x))
```

Output :

```
sin(2*x)/cos(2*x)
```

### 2.21.14 Rewrite tan(x) with sin(2x) and cos(2x) : `tan2sincos2`

`tan2sincos2` takes as argument an expression containing trigonometric functions.
`tan2sincos2` replaces $\tan(x)$ by $\frac{\sin(2.x)}{1+\cos(2.x)}$, in this expression.
Input :

```
tan2sincos2(tan(x))
```

Output :

```
sin(2*x)/(1+cos(2*x))
```

### 2.21.15 Rewrite tan(x) with cos(2x) and sin(2x) : `tan2cossin2`

`tan2cossin2` takes as argument an expression containing trigonometric functions.
`tan2cossin2` replaces $\tan(x)$ by $\dfrac{1-\cos(2.x)}{\sin(2.x)}$ , in this expression.
Input :

```
tan2cossin2(tan(x))
```

Output :

```
(1-cos(2*x))/sin(2*x)
```

### 2.21.16 Rewrite sin, cos, tan in terms of tan(x/2) : `halftan`

`halftan` takes as argument an expression containing trigonometric functions.
`halftan` rewrites $\sin(x),\ \cos(x)$ and $\tan(x)$ in terms of $\tan(\frac{x}{2})$.
Input :

```
halftan(sin(2*x)/(1+cos(2*x)))
```

Output :

```
2*tan(2*x/2)/((tan(2*x/2))^2+1)/
(1+(1-(tan(2*x/2))^2)/((tan(2*x/2))^2+1))
```

Output, after simplification with `normal(ans())` :

```
tan(x)
```

Input :

```
halftan(sin(x)^2+cos(x)^2)
```

Output :

```
(2*tan(x/2)/((tan(x/2))^2+1))^2+
((1-(tan(x/2))^2)/((tan(x/2))^2+1))^2
```

Output, after simplification avec `normal(ans())` :

```
1
```

### 2.21.17 Rewrite trigonometric functions as function of tan(x/2) and hyperbolic functions as function of exp(x): `halftan_hyp2exp`

`halftan_hyp2exp` takes as argument a trigonometric and hyperbolic expression.
`halftan_hyp2exp` rewrites $\sin(x),\ \cos(x), \tan(x)$ in terms of $\tan(\frac{x}{2})$ and $\sinh(x),\ \cosh(x), \tanh(x)$ in terms of $\exp(x)$.
Input :

```
halftan_hyp2exp(tan(x)+tanh(x))
```

Output :

```
(2*tan(x/2))/((1-(tan(x/2))^2))+(((exp(x))^2-1))/
                (((exp(x))^2+1))
```

Input :

```
halftan_hyp2exp(sin(x)^2+cos(x)^2-sinh(x)^2+cosh(x)^2)
```

Output, after simplification with `normal(ans())` :

```
2
```

### 2.21.18 Transform inverse trigonometric functions into logarithms : `atrig2ln`

`atrig2ln` takes as argument an expression containing inverse trigonometric functions.
`atrig2ln` rewrites these functions with complex logarithms.
Input :

```
atrig2ln(asin(x))
```

Output :

```
i*log(x+sqrt(x^2-1))+pi/2
```

### 2.21.19 Transform trigonometric functions into complex exponentials : `trig2exp`

`trig2exp` takes as argument an expression containing trigonometric functions.
`trig2exp` rewrites the trigonometric functions with complex exponentials (WITHOUT linearization).
Input :

```
trig2exp(tan(x))
```

Output :

```
((exp((i)*x))^2-1)/((i)*((exp((i)*x))^2+1))
```

Input :

```
trig2exp(sin(x))
```

Output :

```
(exp((i)*x)-1/(exp((i)*x)))/(2*i)
```

### 2.21.20 Simplify and express preferentially with sine : `trigsin`

`trigsin` takes as argument an expression containing trigonometric functions.
`trigsin` simplify this expression with the formula :
$\sin(x)2 + \cos(x)2 = 1$, $\tan(x) = \frac{\sin(x)}{\cos(x)}$ and tries to rewrite the expression only with sine.
Input :

```
trigsin(sin(x)^4+cos(x)^2+1)
```

Output :

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

### 2.21.21 Simplify and express preferentially with cosine : `trigcos`

`trigcos` takes as argument an expression containing trigonometric functions.
`trigcos` simplifies this expression with the formula :
$\sin(x)2 + \cos(x)2 = 1$, $\tan(x) = \frac{\sin(x)}{\cos(x)}$ and tries to rewrite the expression only with cosine.
Input :

```
trigcos(sin(x)^4+cos(x)^2+1)
```

Output :

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

### 2.21.22 Simplify and express preferentially with tangents : `trigtan`

`trigtan` takes as argument an expression containing trigonometric functions.
`trigtan` simplifies this expression with the formula :
$\sin(x)2 + \cos(x)2 = 1$, $\tan(x) = \frac{\sin(x)}{\cos(x)}$ and tries to rewrite the expression only with tangents.
Input :

```
trigtan(sin(x)^4+cos(x)^2+1)
```

Output :

```
((tan(x))^2/(1+(tan(x))^2))^2+1/(1+(tan(x)^2)+1
```

Output, after simplification with `normal` :

```
(2*tan(x)^4+3*tan(x)^2+2)/(tan(x)^4+2*tan(x))^2+1)
```

### 2.21.23 Rewrite an expression with different options : `convert convertir`

`convert` takes two arguments an expression and an option.
`convert` rewrites this expression applying rules depending on the option. Valid options are :

- `sin` converts an expression like `trigsin`.
- `cos` converts an expression like`trigcos`.
- `sincos` converts an expression like `sincos`.
- `trig` converts an expression like `sincos`.
- `tan` converts an expression like `halftan`.
- `exp` converts an expression like `trig2exp`.
- `ln` converts an expression like `trig2exp`.
- `expln` converts an expression like `trig2exp`.
- `string` converts an expression into a string.
- `matrix` converts a list of lists into a matrix.
- `polynom` converts a Taylor series into a un polynomial by removing the remainder (cf 2.24.22).
- `parfrac` or `partfrac` or `fullparfrac` converts a rational fraction into it's partial fraction decomposition (2.28.9).

`convert` can also :

- convert units, for example `convert(1000_g,_kg)=1.0_kg` (cf 5.1.4).
- write a real as a continued fraction : `convert(a,confrac,'fc')` writes `a` as a continued fraction stored in `fc`. Do not forget to quote the last argument if it was assigned.
  For example, `convert(1.2,confrac,'fc')=[1,5]` and `fc` contains the continue fraction équal to 1.2 (cf 2.8.7).
- transform an integer into the list of its ciphers in a basis, begining with the unit cipher (and reciprocally)
  - `convert(n,base,b)` transforms the integer `n` into the list of its ciphers in a basis `b` begining with the unit cipher.
    For example, `convert(123,base,10)=[3,2,1]` and reciproqualy
  - `convert(l,base,b)` transforms the list `l` into the integer `n` which have `l` as list of its ciphers in a basis `b` begining with the unit cipher.
    For example, `convert([3,2,1],base,10)=123` (cf 2.5).

## 2.22 Fourier transformation

### 2.22.1 Fourier coefficients : `fourier_an` and `fourier_bn` or `fourier_cn`

Let $f$ be a $T$-periodic continuous functions on $\mathbb{R}$ except maybe at a finite number of points. One can prove that if $f$ is continuous at $x$, then;

$$\begin{aligned} f(x) &= \frac{a_0}{2} + \sum_{n=1}^{+\infty} a_n \cos(\frac{2\pi nx}{T}) + b_n \sin(\frac{2\pi nx}{T}) \\ &= \sum_{n=-\infty}^{+\infty} c_n e^{\frac{2i\pi nx}{T}} \end{aligned}$$

where the coefficients $a_n,\ b_n, n \in N$, (or $c_n, n \in Z$) are the Fourier coefficients of $f$. The commands`fourier_an` and `fourier_bn` or `fourier_cn` compute these coefficients.

`fourier_an`

`fourier_an` takes four or five arguments : an expression *expr* depending of a variable, the name of this variable (for example $x$), the period $T$, an integer $n$ and a real $a$ (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 $f(x) = expr$ and such that $f$ is periodic of period $T$:

$$a_n = \frac{2}{T}\int_a^{a+T} f(x)\cos(\frac{2\pi nx}{T})dx$$

To simplify the computations, ons should input `assume(n,integer)` before calling `fourier_an` to specify that $n$ is an integer.
**Example** Let the function $f$, of period $T = 2$, 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 :

```
1/3
```

Input, to have the coefficient $a_n$ ($n \neq 0$) :

```
assume(n,integer);fourier_an(x^2,x,2,n,-1)
```

Output :

```
4*(-1)^n/(pi^2*n^2)
```

`fourier_bn`

`fourier_bn` takes four or five arguments : an expression *expr* depending of a variable, the name of this variable (for example $x$), the period $T$, an integer $n$ and a real $a$ (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 $f(x) = expr$ and periodic of period $T$:

$$b_n = \frac{2}{T}\int_a^{a+T} f(x)\sin(\frac{2\pi nx}{T})dx$$

To simplify the computations, one should input `assume(n,integer)` before calling `fourier_bn` to specify that $n$ is an integer.
**Examples**

- Let the function $f$, of period $T = 2$, defined on $[-1; 1[$ by $f(x) = x^2$.
  Input, to have the coefficient $b_n$ ($n \neq 0$) :

  ```
  assume(n,integer);fourier_bn(x^2,x,2,n,-1)
  ```

  Output :

  ```
  0
  ```

- Let the function $f$, of period $T = 2$, defined on $[-1; 1[$ by $f(x) = x^3$.
  Input, to have the coefficient $b_1$ :

  ```
  fourier_bn(x^3,x,2,1,-1)
  ```

  Output :

  ```
  (2*pi^2-12)/pi^3
  ```

`fourier_cn`

`fourier_cn` takes four or five arguments : an expression *expr* depending of a variable, the name of this variable (for example $x$), the period $T$, an integer $n$ and a real $a$ (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 $f(x) = expr$ and periodic of period $T$:

$$c_n = \frac{1}{T}\int_a^{a+T} f(x)e^{\frac{-2i\pi nx}{T}}dx$$

To simplify the computations, one should input `assume(n,integer)` 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 have $c_0$ :

  ```
  fourier_cn(x^2,x,2,0,-1)
  ```

  Output:

```
1/3
```

Input, to have $c_n$ :

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

```
fourier_cn(x^2,x,2,n,-1)
```

Output:

```
2*(-1)^n/(pi^2*n^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:

  ```
  4/3
  ```

  Input, to have $c_n$ :

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

  ```
  fourier_cn(x^2,x,2,n)
  ```

  Output:

  ```
  ((2*i)*pi*n+2)/(pi^2*n^2)
  ```

- Find the Fourier coefficients $c_n$ of the periodic function $f$ of period $2.\pi$ and defined on $[0;2.\pi[$ by $f(x) = x^2$.
  Input :

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

  ```
  fourier_cn(x^2,x,2*pi,n)
  ```

  Output :

  ```
  ((2*i)*pi*n+2)/n^2
  ```

  If you don't specify `assume(n,integer)`, the output will not be simplified :

```
((2*i)*pi^2*n^2*exp((-i)*n*2*pi)+2*pi*n*exp((-i)*n*2*pi)+
            (-i)*exp((-i)*n*2*pi)+i)/(pi*n^3)
```

You might simplify this expression by replacing `exp((-i)*n*2*pi)` by `1`, input :

```
subst(ans(),exp((-i)*n*2*pi)=1)
```

Output :

```
((2*i)*pi^2*n^2+2*pi*n+-i+i)/pi/n^3
```

This expression is then simplified with `normal`, the final output is :

```
((2*i)*pi*n+2)/n^2
```

Hence for $n \neq 0$, $c_n = \dfrac{2in\pi + 2}{n^2}$. As shown in this example, it is better to input `assume(n,integer)` before calling `fourier_cn`.
We must also compute $c_n$ for $n = 0$, input :

```
fourier_cn(x^2,x,2*pi,0)
```

Output :

```
4*pi^2/3
```

Hence for $n = 0$, $c_0 = \dfrac{4.\pi^2}{3}$.

**Remarks** :

- Input `purge(n)` to remove the hypothesis done on $n$.
- Input `about(n)` or `assume(n)`, to know the hypothesis done on the variable $n$.

### 2.22.2 Discrete Fourier Transform

Let $N$ be an integer. 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 $\omega_N$. If the DFT is defined on 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 $x = [x_0, x_1, ...x_{N-1}]$ then $F_N(x) = y$ is a periodic sequence of period $N$, defined by:

$$(F_{N,\omega_N}(x))_k = y_k = \sum_{j=0}^{N-1} x_j \omega_N^{-k \cdot j}, k = 0..N-1$$

where $\omega_N$ is a primitive $N$-th root of unity. The discrete Fourier transform may be computed faster than by computing each $y_k$ individually, by the Fast Fourier Transform (FFT). `Xcas` implements the FFT algorithm to compute the discrete Fourier transform only if $N$ is a power of 2.

**The properties of the Discrete Fourier Transform**

The Discrete Fourier Transform $F_N$ is a bijective transformation on periodic sequences such that

$$\begin{aligned} F_{N,\omega_N}^{-1} &= \frac{1}{N} F_{N,\omega_N^{-1}} \\ &= \frac{1}{N} \overline{F_N} \quad \text{on } \mathbb{C} \end{aligned}$$

i.e. :

$$(F_N^{-1}(x))_k = \frac{1}{N} \sum_{j=0}^{N-1} x_j \omega_N^{k \cdot j}$$

Inside `Xcas` the discrete Fourier transform and it's inverse are denote by `fft` and `ifft`:

`fft(x)`=$F_N(x)$, `ifft(x)`=$F_N^{-1}(x)$

**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** :

$$\begin{aligned} N * F_N(x \cdot y) &= F_N(x) * F_N(y) \\ F_N(x * y) &= F_N(x) \cdot F_N(y) \end{aligned}$$

**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 $c := [c_0, c_1, ..c_{N-1}]$, where zeroes may be added so that $N$ is a power of 2.

   - Compute the values of $P(x)$ at

   $$x = a_k = \omega_N^{-k} = \exp(\frac{-2ik\pi}{N}), \quad k = 0..N-1$$

   This is just the discrete Fourier transform of $c$ since

   $$P(a_k) = \sum_{j=0}^{N-1} c_j (\omega_N^{-k})^j = F_N(c)_k$$

   Input, for example :

```
P(x):=x+x^2; w:=i
```

Here the coefficients of $P$ are [0,1,1,0], $N = 4$ and $\omega = \exp(2i\pi/4) = i$.
Input :
```
fft([0,1,1,0])
```
Output :
```
[2,-1-i,0,-1+i]
```
hence

- `P(0)=2`,
- `P(-i)=P(w^-1)=-1-i`,
- `P(-1)=P(w^-2)=0`,
- `P(i)=P(w^-3)=-1+i`.

• Compute the values of $P(x)$ at

$$x = b_k = \omega_N^k = \exp(\frac{2ik\pi}{N}), \quad k = 0..N-1$$

This is $N$ times the inverse fourier transform of $c$ since

$$P(a_k) = \sum_{j=0}^{N-1} c_j(\omega_N^k)^j = NF_N^{-1}(c)_k$$

Input, for example :
`P(x):=x+x^2` and `w:=i`
Hence, the coefficients of $P$ are [0,1,1,0], $N = 4$ and $\omega = \exp(2i\pi/4) = i$.
Input :
```
4*ifft([0,1,1,0])
```
Output :
```
[2,-1+i,0,-1-i]
```
hence :

- `P(0)=2`,
- `P(i)=P(w^1)=-1+i`,
- `P(-1)=P(w^2)=0`,
- `P(-i)=P(w^3)=-1-i`.

We find of course the same values as above...

2. Trigonometric interpolation
Let $f$ be periodic function of period $2\pi$, assume that $f(2k\pi/N) = f_k$ 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

$$p(x) = \sum_{j=0}^{N-1} p_j \exp(ijx), \quad p(x_k) = f_k$$

Replacing $x_k$ by it's 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, $p_{-k} = \overline{p}_k$, hence depending whether $N$ is even or odd:

$$\begin{aligned} p(x) &= p_0 + 2\Re(\sum_{k=0}^{\frac{N}{2}-1} p_k \exp(ikx)) + \Re(p_{\frac{N}{2}} \exp(i\frac{Nx}{2})) \\ p(x) &= p_0 + 2\Re(\sum_{k=0}^{\frac{N-1}{2}} p_k \exp(ikx)) \end{aligned}$$

3. Fourier series
Let $f$ be a periodic function of period $2\pi$, such that

$$f(x_k) = y_k, \quad x_k = \frac{2k\pi}{N}, k = 0..N-1$$

Suppose that the Fourier serie 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 $\int_0^{2\pi} f(t)\exp(-int)dt$ may be computed by the trapezoidal rule (note that the Romberg algorithm would not work here, because the Euler Mac Laurin developpement has it's coefficients equal to zero, since the integrated function is periodic, hence all it's derivatives have the same value at 0 and at $2\pi$). 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 :

$$[\tilde{c}_0, ..\tilde{c}_{\frac{N}{2}-1}, \tilde{c}_{\frac{-N}{2}}, ..c_{-1}] = \frac{1}{N} F_N([y_0, y_1 ... y_{(N-1)}])$$

since

- if $n \geq 0$, $\tilde{c_n} = y_n$
- if $n < 0$ $\tilde{c_n} = y_{n+N}$
- $\omega_N = \exp(\frac{2i\pi}{N})$, then $\omega_N^n = \omega_N^{n+N}$

**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 $P$ of degree $m \leq \frac{N}{2}$, then
$$f(t) = P(t) = \sum_{k=-m}^{m-1} c_k \exp(2ik\pi t)$$
the trigonometric polynomial that interpolate $f = P$ is $P$, the numeric approximation of the coefficients are in fact exact ($\tilde{c_n} = c_n$).
- More generally, we can compute $\tilde{c_n} - c_n$.
Suppose that $f$ is equal to it's Fourier serie, i.e. that :
$$f(t) = \sum_{m=-\infty}^{+\infty} c_m \exp(2i\pi m t), \qquad \sum_{m=-\infty}^{+\infty} |c_m| < \infty$$
Then :
$$f(x_k) = f(\frac{2k\pi}{N}) = y_k = \sum_{m=-\infty}^{+\infty} c_m \omega_N^{km}, \quad \tilde{c_n} = \frac{1}{N}\sum_{k=0}^{N-1} y_k \omega_N^{-kn}$$
Replace $y_k$ by it's 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 $m \neq n \pmod N$, $\omega_N^{m-n}$ is a $N$-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, we get
$$\begin{aligned}\tilde{c_n} &= \frac{1}{N}\sum_{m=-\infty}^{+\infty} c_m \sum_{k=0}^{N-1}\omega_N^{k(m-n)} \\ &= \sum_{l=-\infty}^{+\infty} c_{(n+l\cdot N)} \\ &= ...c_{n-2\cdot N} + c_{n-N} + c_n + c_{n+N} + c_{n+2\cdot N} + .....\end{aligned}$$
Conclusion: if $|n| < N/2$, $\tilde{c_n} - c_n$ is a sum of $c_j$ of 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 :

$$\mathtt{x} = \{2, (\sqrt{2})/2, -1, -((\sqrt{2})/2), 0, -((\sqrt{2})/2), -1, (\sqrt{2})/2\}$$

```
fft(x)=[0.0,4.0,4.0,0.0,0.0,0.0,4.0,4.0]
```

After a division by $N = 8$, we get

$$c_0 = 0, c_1 = 4.0/8, c_2 = 4.0/2, c_3 = 0.0,$$
$$c_{-4} = 0, c_{-3} = 0, c_{-2} = 4.0/8, = c_{-1} = 4.0/8$$

Hence $b_k = 0$ and $a_k = c_{-k} + c_k$ is equal to 1 if $k = 1, 2$ 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 vector of their coefficients $a = [a_0, a_1, ..a_{n-1}]$ and $b = [b_0, b_1, ..b_{m-1}]$, we may 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 choosen such that $N = n + m + p$ is a power of 2. If $a = [a_0, a_1, ..a_{n-1}, 0..0]$ and $b = [b_0, b_1, ..b_{m-1}, 0..0]$, then:

$$P(x)Q(x) = \sum_{j=0}^{n+m-1} (a * b)_j x^j$$

We compute $F_N(a)$, $F_N(b)$, then $ab = F_N^{-1}(F_N(a) \cdot F_N(b))$ using the properties

$$NF_N(x \cdot y) = F_N(x) * F_N(y), \quad F_N(x * y) = F_N(x) \cdot F_N(y)$$

### 2.22.3 Fast Fourier Transform : `fft`

`fft` takes as argument a list (or a sequence) $[\mathtt{a}_0, ..\mathtt{a}_{\mathtt{N}-1}]$ where `N` is a power of two. `fft` returns the list $[\mathtt{b}_0, ..\mathtt{b}_{\mathtt{N}-1}]$ such that, for `k=0..N-1`

$$\mathtt{fft}([\mathtt{a}_0, ..\mathtt{a}_{\mathtt{N}-1}])[\mathtt{k}] = \mathtt{b}_\mathtt{k} = \sum_{\mathtt{j}=0}^{\mathtt{N}-1} \mathtt{x}_\mathtt{j} \omega_\mathtt{N}^{-\mathtt{k}\cdot\mathtt{j}}$$

where $\omega_N$ is a primitive $N$-th root of the unity.
Input :

```
fft(0,1,1,0)
```

Output :

```
[2.0, -1-i, 0.0, -1+i]
```

### 2.22.4 Inverse Fast Fourier Transform : `ifft`

`ifft` takes as argument a list $[\mathtt{b}_0, ..\mathtt{b}_{\mathtt{N}-1}]$ where `N` is a power of two.
`ifft` returns the list $[\mathtt{a}_0, ..\mathtt{a}_{\mathtt{N}-1}]$ such that

$$\mathtt{fft}([\mathtt{a}_0, ..\mathtt{a}_{\mathtt{N}-1}]) = [\mathtt{b}_0, ..\mathtt{b}_{\mathtt{N}-1}]$$

Input :

```
ifft([2,-1-i,0,-1+i])
```

Output :

```
[0.0, 1.0, 1.0, 0.0]
```

### 2.22.5 An exercice with `fft`

Here are the temperatures $T$, in Celsius degree, at time $t$ :

| 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

$$p(t) = \frac{1}{2}p_{-m}(\exp(-2i\frac{\pi mt}{24}) + \exp(2i\frac{\pi mt}{24})) + \sum_{k=-m+1}^{m-1} p_k \exp(2i\frac{\pi kt}{24})$$

and

$$p_k = \frac{1}{N}\sum_{k=j}^{N-1} T_k \exp(2i\frac{\pi k}{N})$$

Input :
```
q:=1/8*fft([11,10,17,24,32,26,23,19])
```
Output :
```
q:=[20.25,-4.48115530061+1.72227182413*i,-0.375+0.875*i,
-0.768844699385+0.222271824132*i,0.5,
-0.768844699385-0.222271824132*i,
-0.375-0.875*i,-4.48115530061-1.72227182413*i]
```
hence:

- $p_0 = 20.25$
- $p_1 = -4.48115530061 + 1.72227182413 * i = \overline{p_{-1}}$,
- $p_2 = 0.375 + 0.875 * i = \overline{p_{-2}}$,
- $p_3 = -0.768844699385 + 0.222271824132 * i = \overline{p_{-3}}$,
- $p_{-4} = 0.5$

Indeed

$$q = [q_0, ...q_{N-1}] = [p_0, ..p_{\frac{N}{2}-1}, p_{-\frac{N}{2}}, .., p_{-1}] = \frac{1}{N}F_N([y_0, ..y_{N-1}]) = \frac{1}{\mathtt{N}}\mathtt{fft}(\mathtt{y})$$

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 $k = -4...3$. It remains to compute the value of the interpolation polynomial at point $t0 = 13,75 = 55/4$, 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 Celsius degrees.
Input :

```
q1:=[q[4]/2,q[3],q[2],q[1],q[0]/2]
a:=t0(1) (ou a:=-exp(i*pi*7/48))
g(x):=r2e(q1,x)
evalf(2*re(g(a)))
```

or

```
2.0*re(q[0]/2+q[1]*t0(1)+q[2]*t0(2)+q[3]*t0(3)+q[4]/2*t0(4))
```

Output :

```
29.4863181684
```

**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)
```

Output :

$$\frac{8632428959}{286654464} \simeq 30.1144061688$$

## 2.23 Exponentials and Logarithms

### 2.23.1 Rewrite hyperbolic functions as exponentials : `hyp2exp`

`hyp2exp` takes as argument an hyperbolic expression.
`hyp2exp` rewrites each hyperbolic functions with exponentials (as a rational fraction of one exponential, i.e. WITHOUT linearization).
Input :

```
hyp2exp(sinh(x))
```

Output :

```
(exp(x)-1/(exp(x)))/2
```

### 2.23.2 Expand exponentials : `expexpand`

`expexpand` takes as argument an expression with exponentials.
`expexpand` expands this expression (rewrites exp of sums as product of exp).
Input :

```
expexpand(exp(3*x)+exp(2*x+2))
```

Output :

```
exp(x)^3+exp(x)^2*exp(2)
```

### 2.23.3 Expand logarithms : `lnexpand`

`lnexpand` takes as argument an expression with logarithms.
`lnexpand` expands this expression (rewrites ln of products as sum of ln).
Input :

```
lnexpand(ln(3*x^2)+ln(2*x+2))
```

Output :

```
ln(3)+2*ln(x)+ln(2)+ln(x+1)
```

### 2.23.4 Linearize exponentials : `lin`

`lin` takes as argument an expression with exponentials.
`lin` rewrites hyperbolic functions as exponentials if required, then linearizes this expression (i.e. replace product of exponentials by exponential of sums).
**Examples**

- Input :

  ```
  lin(sinh(x)^2)
  ```

  Output :

  ```
  1/4*exp(2*x)+1/-2+1/4*exp(-(2*x))
  ```

- Input :

  ```
  lin((exp(x)+1)^3)
  ```

  Output :

  ```
  exp(3*x)+3*exp(2*x)+3*exp(x)+1
  ```

### 2.23.5 Collect logarithms : `lncollect`

`lncollect` takes as argument an expression with logarithms.
`lncollect` collects the logarithms (rewrites sum of ln as ln of products). It may be a good idea to factor the expression with `factor` before collecting by `lncollect`).
Input :

```
lncollect(ln(x+1)+ln(x-1))
```

Output :

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

Input :

```
lncollect(exp(ln(x+1)+ln(x-1)))
```

Output :

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

**Warning!!!** For `Xcas`, `log=ln` (use `log10` for 10-base logarithm).

### 2.23.6 Expand powers : `powexpand`

`powexpand` rewrites a power of sum as a product of powers.
Input :

```
powexpand(a^(x+y))
```

Output :

```
a^x*a^y
```

### 2.23.7 Rewrite a power as an exponential : `pow2exp`

`pow2exp` rewrites a power as an exponential.
Input :

```
pow2exp(a^(x+y))
```

Output :

```
exp((x+y)*ln(a))
```

### 2.23.8 Rewrite exp(n*ln(x)) as a power : `exp2pow`

`exp2pow` rewrites expression of the form $\exp(n * \ln(x))$ as a power of $x$.
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)
```

### 2.23.9 Simplify complex exponentials : `tsimplify`

`tsimplify` 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.
Input :

```
tsimplify((sin(7*x)+sin(3*x))/sin(5*x))
```

Output :

```
((exp((i)*x))^4+1)/(exp((i)*x))^2
```

## 2.24 Polynomials

A polynomial of one variable is represented either by a symbolic expression or by the list of it's coefficients by decreasing powers order (dense representation). In the latter case, to avoid confusion with other kinds of list

- use `poly1[...]` as delimiters in inputs
- check for [] [] in `Xcas` output.

Note that polynomials represented as lists of coefficients are always written in decreassing powers order even if `increasing power` is checked in `cas` configuration.

A polynomial of several variables is represented

- by a symbolic expression
- or by a dense recursive 1-d representation like above
- or by a sum of monomials with non-zero coefficients (distributed sparse representation).
  A monomial with several variables is represented by a coefficent and a list of integer (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]`).

### 2.24.1 Convert to a symbolic polynomial : `r2e poly2symb`

`r2e` or `poly2symb` takes as argument

- a list of coefficients of a polynomial (by decreasing order) and a symbolic variable name (by default `x`)
- or a sum of monomials `%%%{coeff,[n1,....nk] %%%}` and a vector of symbolic variables `[x1,...,xk]`).

`r2e` or `poly2symb` transforms the argument into a symbolic polynomial.
Example with univariate polynomials, input :

```
r2e([1,0,-1],x)
```

or :

```
r2e([1,0,-1])
```

or :

```
poly2symb([1,0,-1],x)
```

Output :

```
x*x-1
```

Example with sparse multivariate polynomials, 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-x*y+2*y
```

### 2.24.2 Convert from a symbolic polynomial : `e2r symb2poly`

`e2r` or `symb2poly` takes as argument a symbolic polynomial and either a symbolic variable name (by default `x`) or a list of symbolic variable names.
`e2r` or `symb2poly` transforms the polynomial into a list (dense representation of the univariate polynomial, coefficients written by decreasing order) or into a sum of monomials (sparse representation of multivariate polynomials).
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]%%%}
```

### 2.24.3 Coefficients of a polynomial: `coeff coeffs`

`coeff` or `coeffs` takes three arguments : the polynomial, the name of the variable (or the list of the names of variables) and the degree (or the list of the degrees of the variables).
`coeff` or `coeffs` returns the coefficient of the polynomial of the degree given as third argument. If no degree was specified, `coeffs` return the list of the coefficients of the polynomial, including 0 in the univariate dense case and excluding 0 in the multivariate sparse case.
Input :

```
coeff(-x^4+3*x*y^2+x,x,1)
```

Output :

```
3*y^2+1
```

Input :

```
coeff(-x^4+3x*y^2+x,y,2)
```

Output :

```
3*x
```

Input :

```
coeff(-x^4+3x*y^2+x,[x,y],[1,2])
```

Output :

```
3
```

### 2.24.4 Polynomial degree : `degree`

`degree` takes as argument a polynomial given by it's symbolic representation or by the list of it's coefficients.
`degree` returns the degree of this polynomial (highest degree of it's non-zero monomials).
Input :

```
degree(x^3+x)
```

Output :

```
3
```

Input :

```
degree([1,0,1,0])
```

Output :

```
3
```

### 2.24.5 Polynomial valuation : `valuation ldegree`

`valuation` or `ldegree` takes as argument a polynomial given by a symbolic expression or by the list of it's coefficients.
`valuation` or `ldegree` returns the valuation of this polynomial, that is the lowest degree of it's non-zero monomials.
Input :

```
valuation(x^3+x)
```

Output :

```
1
```

Input :

```
valuation([1,0,1,0])
```

Output :

```
1
```

### 2.24.6 Leading coefficient of a polynomial : `lcoeff`

`lcoeff` takes as argument a polynomial given by a symbolic expression or by the list of it's coefficients.
`lcoeff` returns the leading coefficient of this polynomial, that is the coefficient of the monomial of highest degree.
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 :

```
5*x
```

### 2.24.7 Trailing coefficient degree of a polynomial : `tcoeff`

`tcoeff` takes as argument a polynomial given by a symbolic expression or by the list of its coefficients.
`tcoeff` returns the coefficient of the monomial of lowest degree of this polynomial (`tcoeff`=trailing coefficient).
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 :

```
3*x^2
```

### 2.24.8 Evaluation of a polynomial : `peval polyEval`

`peval` or `polyEval` takes as argument a polynomial `p` given by the list of its coefficients and a real `a` .
`peval` or `polyEval` returns the exact or numeric value of `p(a)` using Horner algorithm.
Input :

```
peval([1,0,-1],sqrt(2))
```

Output :

```
sqrt(2)*sqrt(2)-1
```

Then :

```
normal(sqrt(2)*sqrt(2)-1)
```

Output :

```
1
```

Input :

```
peval([1,0,-1],1.4)
```

Output :

```
0.96
```

### 2.24.9 Factorize $x^n$ in a polynomial : `factor_xn`

`factor_xn` takes as argument a polynomial `P`.
`factor_xn` returns the polynomial `P` written as the product of it's monomial of largest degree $x^n$ (`n=degree(P)`) with a rational fraction having a non-zero finite limit at infinity.
Input :

```
factor_xn(-x^4+3)
```

Output :

```
x^4*(-1+3*x^-4)
```

### 2.24.10 GCD of the coefficients of a polynomial : `content`

`content` takes as argument a polynomial `P` given by a symbolic expression or by the list of it's coefficients.
`content` returns the content of `P`, that is the GCD (greatest common divisor) of the coefficients of `P`.
Input :

```
content(6*x^2-3*x+9)
```

or:

```
content([6,-3,9],x))
```

Output :

```
3
```

### 2.24.11 Primitive part of a polynomial : `primpart`

`primpart` takes as argument a polynomial `P` given by a symbolic expression or by the list of it's coefficients.
`primpart` returns the primitive part of `P`, that is `P` divided by the GCD (greatest common divisor) of it's coefficients.
Input :

```
primpart(6x^2-3x+9)
```

or:

```
primpart([6,-3,9],x))
```

Output :

```
2*x^2-x+3
```

### 2.24.12 Factorization : `collect`

`collect` takes as argument a polynomial or a list of polynomials and optionnaly an algebraic extension like `sqrt(n)` (for $\sqrt{n}$).
`collect` factorizes the polynomial (or the polynomials of the list) on the field of it's coefficient (for example $\mathbb{Q}$) or on the smallest extension containing the optional second argument (e.g. $\mathbb{Q}[\sqrt{n}]$). In complex mode, the field is complexified.
**Examples** :

- Factorize $x2 - 4$ over the integers, input :

  ```
  collect(x^2-4)
  ```

  Output in real mode :

  ```
  (x-2)*(x+2)
  ```

- Factorize $x2 + 4$ over the integers, input :

  ```
  collect(x^2+4)
  ```

  Output in real mode :

  ```
  x^2+4
  ```

  Output in complex mode :

  ```
  (x+2*i)*(x-2*i)
  ```

- Factorize $x2 - 2$ over the integers, input :

  ```
  collect(x^2-2)
  ```

  Output in real mode :

```
x^2-2
```

But if you input :

```
collect(sqrt(2)*(x^2-2))
```

Output :

```
sqrt(2)*(x-sqrt(2))*(x+sqrt(2))
```

- Factorize over the integers :

  $$x3 - 2x2 + 1 \text{ and } x2 - x$$

  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 :

  ```
  ((19*sqrt(5)-10)*((sqrt(5)+15)*x+7*sqrt(5)-5)*
    ((sqrt(5)+25)*x-13*sqrt(5)-15)*(x-1))/6820
  ```

  Or, input :

  ```
  collect(x^3-2*x^2+1,sqrt(5))
  ```

  Output :

  ```
  ((2*sqrt(5)-19)*((sqrt(5)+25)*x-13*sqrt(5)-15)*(-x+1)*((sqrt(5)+15)*x+7*s
  ```

### 2.24.13 Factorization : `factor factoriser`

`factor` takes as argument a polynomial or a list of polynomials and optionnally an algebraic extension, e.g. `sqrt(n)`.
`factor` factorize the polynomial (or the polynomials of the list) on the field of it's coefficients (the field is complexified in complex mode) or on the smallest extension containing the optionnal second argument. Unlike `collect`, `factor` will further factorize each factor of degree 2 if `Sqrt` is checked in the `cas` configuration (see also 2.12.9). You can check the current configuration in the status button under `Xcas` and change the configuration by hitting this status button.
Input :

```
factor(x^2+2*x+1)
```

Output :

```
(x+1)^2
```

Input :

```
factor(x^4-2*x^2+1)
```

Output :

```
(-x+1)^2*(x+1)^2
```

Input :

```
factor(x^ 3-2*x^2+1)
```

Output if `Sqrt` is not checked in the `cas` configuration :

```
(x-1)*(x^2-x-1)
```

Output if `Sqrt` is checked in the `cas` configuration :

```
(x-1)*(x+(sqrt(5)-1)/2)*(x+(-sqrt(5)-1)/2)
```

Input :

```
factor(x^ 3-2*x^2+1,sqrt(5))
```

Output :

```
((2*sqrt(5)-19)*((sqrt(5)+15)*x+7*sqrt(5)-5)*(-x+1)*((sqrt(5)+25)*x-1
```

Input :

```
factor(x^2+1)
```

Output in real mode :

```
x^2+1
```

Output in complex mode :

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

### 2.24.14 Square-free factorization : `sqrfree`

`sqrfree` takes as argument a polynomial.
`sqrfree` factorizes this polynomial as a product of powers of coprime factors, where each factor has roots of multiplicity 1 (in other words, a factor and it's derivative are coprime).
Input :

```
sqrfree((x^2-1)*(x-1)*(x+2))
```

Output :

```
(x^2+3*x+2)*(x-1)^2
```

Input :

```
sqrfree((x^2-1)^2*(x-1)*(x+2)^2)
```

Output :

```
(x^2+3*x+2)*(x-1)^3
```

### 2.24.15 List of factors : `factors`

`factors` has either a polynomial or a list of polynomials as argument.
`factors` returns a list containing the factors of the polynomial and their exponents.
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 :

```
[[x-1,1,x^2-x-1,1],[x,1,x-1,1]]
```

Input :

```
factors([x^2,x^2-1])
```

Output :

```
[[x,2],[x+1,1,x-1,1]]
```

### 2.24.16 Evaluate a polynomial : `horner`

`horner` takes two arguments : a polynomial `P` given by its symbolic expression or by the list of its coefficients and a number `a`.
`horner` returns `P(a)` computed using Hörner algorithm.
Input :

```
horner(x^2-2*x+1,2)
```

or :

```
horner([1,-2,1],2)
```

Output :

```
1
```

### 2.24.17 Rewrite in terms of the powers of (x-a) : `ptayl`

`ptayl` is used to rewrite a polynomial `P` depending of `x` in terms of the powers of `(x-a)` (`ptayl` means polynomial Taylor)
`ptayl` takes two arguments: a polynomial `P` given by a symbolic expression or by the list of it's coefficients and a number `a`.
`ptayl` returns the polynomial `Q` such that `Q(x-a)=P(x)`
Input :

```
ptayl(x^2+2*x+1,2)
```

Output, the polynomial Q:

```
x^2+6*x+9
```

Input :

```
ptayl([1,2,1],2)
```

Output :

```
[1,6,9]
```

**Remark**

```
P(x)=Q(x-a)
```

i.e. for the example :
$x2 + 2x + 1 = (x - 2)2 + 6(x - 2) + 9$

### 2.24.18 Compute with the exact root of a polynomial : `rootof`

Let $P$ and $Q$ be two polynomials given by the list of their coefficients then `rootof(P,Q)` denotes the value $P(\alpha)$ where $\alpha$ 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 degree($P$) <degree($Q$). If the resulting rootof is the solution of a second degree equation, it will be simplified.

**Example**
Let $\alpha$ be the root with largest imaginary part of $Q(x) = x4 + 10x2 + 1$ (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 $\alpha$ by `rootof([1,0],[1,0,10,0,1])`.
  Output :

  ```
  rootof([[-1,0,-10,0],[1,0,10,0,1]])
  ```

  i.e. :
  $$\frac{1}{\alpha} = -\alpha 3 - 10\alpha$$

- Compute $\alpha 2$. Input :

  ```
  normal(rootof([1,0],[1,0,10,0,1])^2)
  ```

  or (since $P(x) = x2$ is represented by [1,0,0]) input

  ```
  normal(rootof([1,0,0],[1,0,10,0,1]))
  ```

  Output :

  ```
  -5-2*sqrt(6)
  ```

### 2.24.19 Exact roots of a polynomial : `roots`

`roots` takes as arguments a symbolic polynomial expression and the name of it's variable.
`roots` returns a 2 columns matrix : each row is the list of a root of the polynomial and it's multiplicity.
**Examples**

- Find the roots of $P(x) = x5 - 2x4 + x3$.
  Input :

  ```
  roots(x^5-2*x^4+x^3)
  ```

  Output :

  ```
  [[8+3*sqrt(7),1],[8-3*sqrt(7),1],[0,3]]
  ```

- Find the roots of $x^{10} - 15x8 + 90x6 - 270x4 + 405x2 - 243 = (x2 - 3)5$.
  Input :

  ```
  roots(x^10-15*x^8+90*x^6-270*x^4+405*x^2-243)
  ```

  Output :

  ```
  [[sqrt(3),5],[-(sqrt(3)),5]]
  ```

- Find the roots of $(t3 - 3)$.
  Input :

  ```
  roots(t^3-1,t)
  ```

  Output :

  ```
  [[(-1+(i)*sqrt(3))/2,1],[(-1-(i)*sqrt(3))/2,1],[1,1]]
  ```

### 2.24.20 Coefficients of a polynomial defined by its roots : `pcoeff pcoef`

`pcoeff` (or `pcoef`) takes as argument a list of the roots of a polynomial $P$.
`pcoeff` (or `pcoef`) returns a univariate polynomial having these roots, represented as the list of it's coefficients by decreasing order.
Input :

```
pcoef([1,2,0,0,3])
```

Output :

```
[1,-6,11,-6,0,0]
```

i.e. $(x-1)(x-2)(x2)(x-3) = x5 - 6x4 + 11x3 - 6x2$.

### 2.24.21 Truncate of order $n$ : `truncate`

`truncate` takes as argument, a polynomial and an integer `n`.
`truncate` truncates this polynomial at order `n` (removing all terms of order greater or equal to `n+1`).
`truncate` may be used to transform a series expansion into a polynomial or to compute series expansion step by step.
Input :

```
truncate((1+x+x^2/2)^3,4)
```

Output :

```
(9*x^4+16*x^3+18*x^2+12*x+4)/4
```

Input :

```
truncate(series(sin(x)),4)
```

Output :

```
(-x^3-(-6)*x)/6
```

Note that the returned polynomial is normalized.

### 2.24.22 Convert a series expansion into a polynomial : `convert convertir`

`convert`, with the option `polynom`, converts a Taylor series into a polynomial.
It should be used for operations like drawing the graph of the Taylor series of a function near a point.
`convert` takes two arguments : an expression and the option `polynom`.
`convert` replaces the `order_size` functions by 0 inside the expression.
Input :

```
convert(taylor(sin(x)),polynom)
```

Output :

```
x+1/-6*x^3+1/120*x^5+x^6*0
```

Input :

```
convert(series(sin(x),x=0,6),polynom)
```

Output :

```
x+1/-6*x^3+1/120*x^5+x^7*0
```

### 2.24.23 Random polynomial : `randpoly randPoly`

`randpoly` (or `randPoly`) takes two arguments: the name of a variable (by default `x`) and an integer `n` (the order of the arguments is not important).
`randpoly` returns a polynomial with respect to the variable given argument (or `x` if none was provided), of degree the second argument, having as coefficients random integers evenly distributed on -99..+99.
Input :

```
randpoly(t,4)
```

Output for example:

```
-8*t^4-87*t^3-52*t^2+94*t+80
```

Input :

```
randpoly(4)
```

Output for example:

```
70*x^4-46*x^3-7*x^2-24*x+52
```

Input :

```
randpoly(4,u)
```

Output for example:

```
2*u^4+33*u^3-6*u^2-92*u-12
```

### 2.24.24 Change the order of variables : `reorder`

`reorder` takes two arguments : an expression and a vector of variable names.
`reorder` expands the expression according to the order of variables given as second argument.
Input :

```
reorder(x^2+2*x*a+a^2+z^2-x*z,[a,x,z])
```

Output :

```
a^2+2*a*x+x^2-x*z+z^2
```

**Warning** :
The variables must be symbolic (if not, purge them before calling `reorder`)

### 2.24.25 Random list : `ranm`

`ranm` takes as argument an integer `n`.
`ranm` 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 2.41.3 and **??**).
Input :

```
ranm(3)
```

Output :

```
[68,-21,56]
```

### 2.24.26 Lagrange's polynomial : `lagrange interp`

`lagrange` takes as argument two lists of size `n` (resp a matrix with two rows and `n` columns) and the name of a variable `var` (by default `x`).
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 $y_k$ ($k = 1..n$).
`lagrange` returns a polynomial expression `P` with respect to `var` of degree `n-1`, such that $P(x_i) = y_i$.
Input :

```
lagrange([[1,3],[0,1]])
```

or :

```
lagrange([1,3],[0,1])
```

Output :

```
(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 :

```
(y-1)/2
```

**Warning**
`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 the Lagrange polynomial would be computed each time `f` is called (indeed in a function definition, the second member of the affectation 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`.

### 2.24.27 Natural splines: `spline`

#### Definition

Let $\sigma_n$ be a subdivision of a real interval $[a, b]$ :

$$a = x_0, \quad x_1, \quad ..., \quad x_n = b$$

$s$ is a spline function of degree $l$, if $s$ is an application from $[a, b]$ in $\mathbb{R}$ such that :

- $s$ has continuous derivatives up to the order $l - 1$,
- on each interval of the subdivision, $s$ is a polynomial of degree less or equal than $l$.

#### Theorem

The set of spline functions of degree $l$ on $\sigma_n$ is a $\mathbb{R}$-vectorial subspace of dimension $n + l$.

**Proof**

On $[a, x_1]$, $s$ is a polynomial $A$ of degree less or equal to $l$, hence on $[a, x_1]$, $s = A(x) = a_0 + a_1x + ...a_lx^l$ and $A$ is a linear combinaison of $1, x, ...x^l$.
On $[x_1, x_2]$, $s$ is a polynomial $B$ of degree less or equal to $l$, hence on $[x_1, x_2]$, $s = B(x) = b_0 + b_1x + ...b_lx^l$.
$s$ has continuous derivatives up to order $l - 1$, hence :

$$\forall 0 \leq j \leq l - 1, \quad B^{(j)}(x_1) - A^{(j)}(x_1) = 0$$

therefore $B(x) - A(x) = \alpha_1(x - x_1)^l$ or $B(x) = A(x) + \alpha_1(x - x_1)^l$.
Define the function :

$$\text{q}_1(x) = \begin{cases} 0 & \text{sur} \quad [a, x_1] \\ (x - x_1)^l & \text{sur} \quad [x_1, b] \end{cases}$$

Hence :

$$s|_{[a,x_2]} = a_0 + a_1x + ...a_lx^l + \alpha_1q_1(x)$$

On $[x_2, x_3]$, $s$ is a polynomial $C$ of degree less or equal than $l$, hence on $[x_2, x_3]$, $s = C(x) = c_0 + c_1x + ...c_lx^l$.
$s$ has continuous derivatives until $l - 1$, hence :

$$\forall 0 \leq j \leq l - 1, \quad C^{(j)}(x_2) - B^{(j)}(x_2) = 0$$

therefore $C(x) - B(x) = \alpha_2(x - x_2)^l$ or $C(x) = B(x) + \alpha_2(x - x_2)^l$.
Define the function :

$$\text{q}_2(x) = \begin{cases} 0 & \text{on} \quad [a, x_2] \\ (x - x_2)^l & \text{on} \quad [x_2, b] \end{cases}$$

Hence : $s|_{[a,x_3]} = a_0 + a_1x + ...a_lx^l + \alpha_1q_1(x) + \alpha_2q_2(x)$
And so on, the functions are defined by :

$$\forall 1 \leq j \leq n - 1, \text{q}_j(x) = \begin{cases} 0 & \text{on} \quad [a, x_j] \\ (x - x_j)^l & \text{on} \quad [x_j, b] \end{cases}$$

hence,

$$s|_{[a,b]} = a_0 + a_1x + ...a_lx^l + \alpha_1q_1(x) + .... + \alpha_{n-1}q_{n-1}(x)$$

and $s$ is a linear combination of $n + l$ independant functions $1, x, ..x^l, q_1, ..q_{n-1}$.

**Interpolation with spline functions**

If we want to interpolate a function $f$ on $\sigma_n$ by a spline function $s$ of degree $l$, then $s$ must verify $s(x_k) = y_k = f(x_k)$ for all $0 \geq k \geq n$. Hence there are $n+1$ conditions, and $l-1$ degrees of liberty. We 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 unicity 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 liberty. 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 liberty. 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

$$\forall 1 \leq j \leq 2m-1, \quad s^{(j)}(a) = s^{(j)}(b)$$

A natural spline is a spline function which verifies the natural interpolation constraints.

`spline` takes as arguments a list of abscissa (by increasing order), a list of ordinates, a variable name, and a degree.
`spline` returns the natural spline function (with the specified degree and crossing points) as a list of polynomials, each polynomial being valid on an interval.

Examples:

1. a 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 is a list of two polynomial expressions of $x$ :

   $$[-5*x3/4+13*x/4+1, \quad 5*(x-1)3/4-15*(x-1)2/4+(x-1)/-2+3]$$

   defined respectivly on the intervals $[0, 1]$ and $[1, 2]$.

2. a 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 is a list of three polynomial functions of $x$ :

   $$[(-62 * x4 + 304 * x)/121 + 1,$$

   $$(201 * (x - 1)4 - 248 * (x - 1)3 - 372 * (x - 1)2 + 56 * (x - 1))/121 + 3,$$

   $$(-139 * (x - 2)4 + 556 * (x - 2)3 + 90 * (x - 2)2 + -628 * (x - 2))/121]$$

   defined respectivly on the intervals $[0, 1]$, $[1, 2]$ and $[2, 3]$.

3. 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 :

   $$[((3 * \pi3 + (-7 * \pi2) * x + 4 * x3) * 1/3)/(\pi3),$$

   $$((15 * \pi3 + (-46 * \pi2) * x + 36 * \pi * x2 - 8 * x3) * 1/12)/(\pi3)]$$

## 2.25 Arithmetic and polynomials

Polynomials are represented by expressions or by list of coefficients by 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 modular coefficients in $\mathbb{Z}/n\mathbb{Z}$, use `% n` for each coefficient of the list or apply it to the expression defining the polynomial.

### 2.25.1 The divisors of a polynomial : `divis`

`divis` takes as argument a polynomial (or a list of polynomials) and returns the list of the divisors of the polynomial(s).
Input :

```
divis(x^4-1)
```

Output :

```
[1,x^2+1,x+1,(x^2+1)*(x+1),x-1,(x^2+1)*(x-1),
(x+1)*(x-1),(x^2+1)*(x+1)*(x-1)]
```

Input :

```
divis([x^2,x^2-1])
```

Output :

```
[[1,x,x^2],[1,x+1,x-1,(x+1)*(x-1)]]
```

### 2.25.2 Euclidean quotient : `quo`

`quo` returns the euclidean quotient $q$ of the Euclidean division between two polynomials (decreasing power division) If the polynomials are represented as expressions, the variable may be specified as a third argument.
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, the quotient of $x^2 + 2x + 4$ by $x^2 + x + 2$ one can also input :

```
quo([1,2,4],[1,1,2])
```

Output :

```
[1]
```

that is to say the polynomial `1`.

### 2.25.3 Euclidean quotient : `Quo`

`Quo` is the inert form of `quo`.
`Quo` returns the euclidean quotient between two polynomials (decreasing power division) without evaluation. It is used when `Xcas` is in Maple mode to compute the euclidean quotient of the division of two polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$ using Maple-like syntax.
In `Xcas` mode, input :

```
Quo(x^2+2*x+1,x)
```

Output :

```
quo(x^2+2*x+1,x)
```

In `Maple` mode, input :

```
Quo(x^3+3*x,2*x^2+6*x+5) mod 5
```

Output :

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

The division was done using modular arithmetic, unlike with

```
quo(x^3+3*x,2*x^2+6*x+5) mod 5
```

where the division is done in $\mathbb{Z}[X]$ and reduced after to:

```
3*x-9
```

If `Xcas` is not in Maple mode, polynomial division in $\mathbb{Z}/p\mathbb{Z}[X]$ is done e.g. by :

```
quo((x^3+3*x)% 5,(2x^2+6x+5)%5)
```

### 2.25.4 Euclidean remainder : `rem`

`rem` returns the euclidean remainder between two polynomials (decreasing power division). If the polynomials are represented as expressions, the variable may be specified as a third argument.
Input :

```
rem(x^3-1,x^2-1)
```

Output :

```
x-1
```

To have the remainder of $x^2 + 2x + 4$ by $x^2 + x + 2$ we can also input :

```
rem([1,2,4],[1,1,2])
```

Output :

```
[1,2]
```

i.e. le polynomial $x + 2$.

### 2.25.5 Euclidien remainder: `Rem`

`Rem` is the inert form of `rem`.
`Rem` returns the euclidean remainder between two polynomials (decreasing power division) without evaluation. It is used when `Xcas` is in Maple mode to compute the euclidean remainder of the division of two polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$ using Maple-like syntax.
In `Xcas` mode, input :

```
Rem(x^3-1,x^2-1)
```

Output :

```
rem(x^3-1,x^2-1)
```

In `Maple` mode, input :

```
Rem(x^3+3*x,2*x^2+6*x+5) mod 5
```

Output :

```
2*x
```

The division was done using modular arithmetic, unlike with

```
rem(x^3+3*x,2*x^2+6*x+5) mod 5
```

where the division is done in $\mathbb{Z}[X]$ and reduced after to:

```
12*x
```

If `Xcas` is not in Maple mode, polynomial division in $\mathbb{Z}/p\mathbb{Z}[X]$ is done e.g. by :

```
rem((x^3+3*x)% 5,(2x^2+6x+5)%5)
```

### 2.25.6 Quotient and remainder : `quorem divide`

`quorem` (or `divide`) returns the list of the quotient and the remainder of the euclidian division (by decreassing power) of two polynomials.
Input :

```
quorem([1,2,4],[1,1,2])
```

Output :

```
[poly1[1],poly1[1,2]]
```

Input :

```
quorem(x^3-1,x^2-1)
```

Output :

```
[x,x-1]
```

### 2.25.7 GCD of two polynomials with Euclide algorithm: `gcd`

`gcd` denotes the gcd (greatest common divisor) of two polynomials (or of a list of polynomials or of a sequence of polynomials) (see also 2.6.2 for GCD of integers).

**Examples**

Input :

```
gcd(x^2+2*x+1,x^2-1)
```

Output :

```
x+1
```

Input :

```
gcd(x^2-2*x+1,x^3-1,x^2-1,x^2+x-2)
```

or

```
gcd([x^2-2*x+1,x^3-1,x^2-1,x^2+x-2])
```

Output :

```
x-1
```

For polynomials with modular coefficients, input e.g. :

```
gcd((x^2+2*x+1) mod 5,(x^2-1) mod 5)
```

Output :

```
x % 5
```

Note that :

```
gcd(x^2+2*x+1,x^2-1) mod 5
```

will output :

```
1
```

since the mod operation is done after the GCD is computed in $\mathbb{Z}[X]$.

### 2.25.8 GCD of two polynomials with Euclide algorithm : `Gcd`

`Gcd` is the inert form of `gcd`. `Gcd` returns the gcd (greatest common divisor) of two polynomials (or of a list of polynomials or of a sequence of polynomials) without evaluation. It is used when `Xcas` is in Maple mode to compute gcd of polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$ using Maple-like syntax.

Input in `Xcas` mode :

```
Gcd(x^3-1,x^2-1)
```

Output :

```
gcd(x^3-1,x^2-1)
```

Input in `Maple` mode :

```
Gcd(x^2+2*x,x^2+6*x+5) mod 5
```

Output :

```
1
```

### 2.25.9 Choosing the GCD algorithm of two polynomials : `ezgcd heugcd modgcd psrgcd`

`ezgcd heugcd modgcd psrgcd` denotes the gcd (greatest common divisor) of two univariate or multivariate polynomials with coefficients in $\mathbb{Z}$ or $\mathbb{Z}[i]$ using a specific algorithm :

- `ezgcd` ezgcd algorithm,
- `heugcd` heuristic gcd algorithm,
- `modgcd` modular algorithm,
- `psrgcd` sub-resultant algorithm.

Input :

```
ezgcd(x^2-2*x*y+y^2-1,x-y)
```

or

```
heugcd(x^2-2*x*y+y^2-1,x-y)
```

or

```
modgcd(x^2-2*x*y+y^2-1,x-y)
```

or

```
psrgcd(x^2-2*x*y+y^2-1,x-y)
```

Output :

```
1
```

Input :

```
ezgcd((x+y-1)*(x+y+1),(x+y+1)^2)
```

or

```
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 successfull 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
```

### 2.25.10 LCM of two polynomials : `lcm`

`lcm` returns the LCM (Least Common Multiple) of two polynomials (or of a list of polynomials or of a sequence of polynomials) (see 2.6.5 for LCM of integers).
Input :

```
lcm(x^2+2*x+1,x^2-1)
```

Output :

```
(x+1)*(x^2-1)
```

Input :

```
lcm(x,x^2+2*x+1,x^2-1)
```

or

```
lcm([x,x^2+2*x+1,x^2-1])
```

Output :

```
(x^2+x)*(x^2-1)
```

### 2.25.11 Bézout's Identity : `egcd gcdex`

This function compute the polynomial coefficients of the Bézout's Identity (also known as Extended Greatest Common Divisor). Given two polynomials $A(x), B(x)$, `egcd` computes 3 polynomials $U(x), V(x)$ and $D(x)$ such that :

$$U(x) * A(x) + V(x) * B(x) = D(x) = GCD(A(x), B(x))$$

`egcd` takes 2 or 3 arguments: the polynomials $A$ and $B$ as expressions in terms of a variable, if the variable is not specified it will default to $x$. Alternatively, $A$ and $B$ may be given as list-polynomials.
Input :

```
egcd(x^2+2*x+1,x^2-1)
```

Output :

```
[1,-1,2*x+2]
```

Input :

```
egcd([1,2,1],[1,0,-1])
```

Output :

```
[[1],[-1],[2,2]]
```

Input :

```
egcd(y^2-2*y+1,y^2-y+2,y)
```

Output :

```
[y-2,-y+3,4]
```

Input :

```
egcd([1,-2,1],[1,-1,2])
```

Output :

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

### 2.25.12 Solving au+bv=c over polynomials: `abcuv`

`abcuv` solves the polynomial equation

$$C(x) = U(x) * A(x) + V(x) * B(x)$$

where $A, B, C$ are given polynomials and $U$ and $V$ are unknown polynomials. $C$ must be a multiple of the gcd of $A$ and $B$ for a solution to exist. `abcuv` takes 3 expressions as argument, and an optional variable specification (which defaults to $x$) and returns a list of 2 expressions ($U$ and $V$). Alternatively, the polynomials $A, B, C$ may be entered as list-polynomials.
Input :

```
abcuv(x^2+2*x+1 ,x^2-1,x+1)
```

Output :

```
[1/2,1/-2]
```

Input :

```
abcuv(x^2+2*x+1 ,x^2-1,x^3+1)
```

Output :

```
[1/2*x^2+1/-2*x+1/2,-1/2*x^2-1/-2*x-1/2]
```

Input :

```
abcuv([1,2,1],[1,0,-1],[1,0,0,1])
```

Output :

```
[poly1[1/2,1/-2,1/2],poly1[1/-2,1/2,1/-2]]
```

### 2.25.13 Chinese remainders : `chinrem`

`chinrem` takes two lists as argument, each list being made of 2 polynomials (either expressions or as a list of coefficients in decreassing order). If the polynomials are expressions, an optionnal third argument may be provided to specify the main variable, by default `x` is used. `chinrem([A,R],[B,Q])` returns the list of two polynomials `P` and `S` such that :

$$S = R.Q, \quad P = A \pmod R, P = B \pmod Q$$

If `R` and `Q` are coprime, a solution `P` always exists and all the solutions are congruent modulo `S=R*Q`. For example, assume we want to solve :

$$\left\{ \begin{array}{lll} P(x) = & x & \bmod (x^2+1) \\ P(x) = & x-1 & \bmod (x^2-1) \end{array} \right.$$

Input :

```
chinrem([[1,0],[1,0,1]],[[1,-1],[1,0,-1]])
```

Output :

```
[[1/-2,1,1/-2],[1,0,0,0,-1]]
```

or :

```
chinrem([x,x^2+1],[x-1,x^2-1])
```

Output :

```
[1/-2*x^2+x+1/-2,x^4-1]
```

hence $P(x) = -\dfrac{x^2-2.x+1}{2} \pmod{x^4-1}$
Another example, 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+2*y^2+y+1]
```

### 2.25.14 Cyclotomic polynomial : `cyclotomic`

`cyclotomic` takes an integer $n$ as argument and returns the list of the coefficients of the cyclotomic polynomial of index $n$. This is the polynomial having the $n$-th pritmitive roots of the unity as zeros (a $n$-th root of the unity is primitive if the set of its powers is the set of all the $n$-th root of the unity).

For example, let $n = 4$, the fourth roots of the unity are: $\{1, i, -1, -i\}$ and the primitive roots are: $\{i, -i\}$. Hence, the cyclotomic polynomial of index $4$ is $(x - i).(x + i) = x^2 + 1$. Verification:

```
cyclotomic(4)
```

Output :

```
[1,0,1]
```

Another example, 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

$$(x^{10} - 1) * (x^2 + 1) * (x^8 - x^6 + x^4 - x^2 + 1) = x^{20} - 1$$

### 2.25.15 Sturm sequences and number of sign changes of $P$ on $]a;\ b]$ : `sturm`

`sturm` takes two or four arguments : $P$ a polynomial expression or $P/Q$ a rational fraction and a variable name or $P$ a polynomial expression, a variable name and two real or complex numbers $a$ and $b$.

If `sturm` takes two arguments, `sturm` returns the list of the Sturm sequences and multiplicities of the square-free factors of $P$ (or $P/Q$) (in this case `sturm` behaves like `sturmseq`).

If `sturm` takes four arguments, it behaves like `sturmab` :

- if $a$ and $b$ are reals, `sturm` returns the number of sign changes of $P$ on $]a;\ b]$
- if $a$ or $b$ are complex, `sturm` returns the number of complex roots of $P$ in the rectangle having $a$ and $b$ as opposite vertices.

Input :

```
sturm(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)
```

Output :

```
[2,[[1,0,0,1],[3,0,0],-9],1,[[1,2],1]]
```

Input :

```
sturm(x^2*(x^3+2),x,-2,0)
```

Output :

```
1
```

### 2.25.16 Number of zeros in $[a, b[$ : `sturmab`

`sturmab` takes four arguments: a polynomial expression $P$, a variable name and two real or complex numbers $a$ and $b$

- if $a$ and $b$ are reals, `sturmab` 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 $G = \gcd(P, \text{diff}(P))$.
- if $a$ or $b$ are complex, `sturmab` returns the number of complex roots of $P$ in the rectangle having $a$ and $b$ as opposite vertices.

Input :

```
sturmab(x^2*(x^3+2),x,-2,0)
```

Output :

```
1
```

Input :

```
sturmab(x^3-1,x,-2-i,5+3i
```

Output :

```
3
```

Input :

```
sturmab(x^3-1,x,-i,5+3i
```

Output :

```
1
```

**Warning !!!!**
$P$ is defined by its symbolic expression.
Input :
`sturmab([1,0,0,2,0,0],x,-2,0),`
Output :
`Bad argument type.`

### 2.25.17 Sturm sequences : `sturmseq`

`sturmseq` takes as argument, a polynomial expression $P$ or a rationnal fraction $P/Q$ and returns the list of the Sturm sequences of the square-free factors of odd multiplicity of $P$ (or of $P/Q$). For $F$ a square-free factor of odd multiplicity, the Sturm sequence $R_1, R_2, ...$ is made from $F$, $F'$ by a recurrence relation :

- $R_1$ is the opposite of the euclidian division remainder of $F$ by $F'$ then,
- $R_2$ is the opposite of the euclidian division remainder of $F'$ by $R_1$,
- ...
- and so on untill $R_k = 0$.

Input :

```
sturmseq(2*x^3+2)
```

or

```
sturmseq(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 :

```
sturmseq((2*x^3+2)/(3*x^2+2),x)
```

Output :

```
[2,[[1,0,0,1],[3,0,0],-9],1,[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 :

```
sturmseq((x^3+1)^2,x)
```

Output :

```
[1,1]
```

Indeed $F = 1$.
Input :

```
sturmseq(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`).
**Warning !!!!**
$P$ is defined by its symbolic expression.
Input :
`sturmseq([1,0,0,1],x),`
Output :
`Bad argument type.`

### 2.25.18 Sylvester matrix of two polynomials : `sylvester`

`sylvester` takes two polynomials as arguments.
`sylvester` returns the Sylvester matrix $S$ of these polynomials.
If $A(x) = \sum_{i=0}^{i=n} a_i x^i$ and $B(x) = \sum_{i=0}^{i=m} b_i x^i$ are 2 polynomials, their Sylvester matrix $S$ is a squared matrix of size `m+n` where `m=degree(B(x))` and `n=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 :

$$\begin{pmatrix} s_{(m+1)1}=b_m & s_{(m+1)2}=b_{m-1} & \cdots & s_{(m+1)(m+1)}=b_0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ s_{(m+n)1}=0 & s_{(m+n)2}=0 & \cdots & s_{(m+n)(m+1)}=b_{n-1} & b_{n-2} & \cdots & b_0 \end{pmatrix}$$

Input :

```
sylvester(x^3-p*x+q,3*x^2-p,x)
```

Output :

```
[[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]]
```

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 :

```
-4*p^3-27*q^2
```

### 2.25.19 Resultant of two polynomials : `resultant`

`resultant` takes as argument two polynomials and returns the resultant of the two polynomials.
The resultant of two polynomials is the determinant of their Sylvester matrix $S$. The Sylvester matrix $S$ of two polynomials $A(x) = \sum_{i=0}^{i=n} a_i x^i$ and $B(x) = \sum_{i=0}^{i=m} b_i x^i$ is a square matrix with $m+n$ rows and columns; its first $m$ rows are made from the coefficients of $A(X)$:

$$\begin{pmatrix} s_{11}=a_n & s_{12}=a_{n-1} & \cdots & s_{1(n+1)}=a_0 & 0 & \cdots & 0 \\ s_{21}=0 & s_{22}=a_n & \cdots & s_{2(n+1)}=a_1 & s_{2(n+2)}=a_0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ s_{m1}=0 & s_{m2}=0 & \cdots & s_{m(n+1)}=a_{m-1} & s_{m(n+2)}=a_{m-2} & \cdots & a_0 \end{pmatrix}$$

and the following $n$ rows are made in the same way from the coefficients of $B(x)$ :

$$\begin{pmatrix} s_{(m+1)1}=b_m & s_{(m+1)2}=b_{m-1} & \cdots & s_{(m+1)(m+1)}=b_0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ s_{(m+n)1}=0 & s_{(m+n)2}=0 & \cdots & s_{(m+n)(m+1)}=b_{n-1} & b_{n-2} & \cdots & b_0 \end{pmatrix}$$

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 degree($U$) <degree($B$) and degree($V$) <degree($A$), and this solution has integer coefficients.

Input :

```
resultant(x^3-p*x+q,3*x^2-p,x)
```

Output :

```
-4*p^3-27*q^2
```

**Remark**
discriminant(P)=resultant(P,P').

**An example using resultant**
Let, $F1$ and $F2$ be 2 fixed points in the plan and $A$, a variable point on the circle of center $F1$ and radius $2a$. Find the cartesian equation of the set of points $M$, intersection of the line $F1A$ and of the segment bisector of $F2A$.

Geometric answer :

$$MF1 + MF2 = MF1 + MA = F1A = 2a$$

hence $M$ is on an ellipse with focus $F1, F2$ and major axis $2a$.

Analytic answer : In the Cartesian coordinate system of center $F1$ and $x$-axis having the same direction than the vector $F1F2$, the coordinates of $A$ are :

$$A = (2a\cos(\theta); 2a\sin(\theta))$$

where $\theta$ is the $(Ox, OA)$ angle. Now 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 if $I$ is the midel point of $AF2$, since the coordinates of $F2$ are $F2 = (2c, 0)$, the coordinates of $I$

$$I = (c + ax/2; ay/2) = (c + a\frac{1-t^2}{1+t^2}; a\frac{2t1-t^2}{1+t^2})$$

$IM$ is orthogonal to $AF2$, hence $M = (x; y)$ verify the equation $eq1 = 0$ where

$$eq1 := (x - ix) * (ax - 2 * c) + (y - iy) * ay$$

But $M = (x; y)$ is also on $F1A$, hence $M$ verify the equation $eq2 = 0$

$$eq2 := y/x - ay/ax$$

The resultant of both equations with respect to $t$ `resultant(eq1,eq2,t)` is a polynomial $eq3$ depending on the variables $x, y$, independant 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 $-64 \cdot (\mathtt{x}^2 + \mathtt{y}^2)$ is always different from zero, hence the locus equation of $M$ :

$$\mathtt{x}^2\mathtt{a}^2 - \mathtt{x}^2\mathtt{c}^2 + -2\mathtt{x}\mathtt{a}^2\mathtt{c} + 2\mathtt{x}\mathtt{c}^3 - \mathtt{a}^4 + 2\mathtt{a}^2\mathtt{c}^2 + \mathtt{a}^2\mathtt{y}^2 - \mathtt{c}^4 = 0$$

If the frame origin is $O$, the middle point of $F1F2$, we find the cartesian equation of an ellipse. To make the change of origin $\overrightarrow{F1M} = \overrightarrow{F1O} + \overrightarrow{OM}$, input :

$$\mathtt{normal(subst(}\mathtt{x}^2 \cdot \mathtt{a}^2 - \mathtt{x}^2 \cdot \mathtt{c}^2 + -2 \cdot \mathtt{x} \cdot \mathtt{a}^2 \cdot \mathtt{c} + 2 \cdot \mathtt{x} \cdot \mathtt{c}^3 - \mathtt{a}^4 + 2 \cdot \mathtt{a}^2 \cdot \mathtt{c}^2 + \mathtt{a}^2 \cdot \mathtt{y}^2 - \mathtt{c}^4, [\mathtt{x},\mathtt{y}] = [\mathtt{c} + \mathtt{X}, \mathtt{Y}]))$$

Output :

$$-\mathtt{c}^2 * \mathtt{X}^2 + \mathtt{c}^2 * \mathtt{a}^2 + \mathtt{X}^2 * \mathtt{a}^2 - \mathtt{a}^4 + \mathtt{a}^2 * \mathtt{Y}^2$$

or if $b^2 = a^2 - c^2$, input :

$$\mathtt{normal(subst(}-\mathtt{c}^2 * \mathtt{X}^2 + \mathtt{c}^2 * \mathtt{a}^2 + \mathtt{X}^2 * \mathtt{a}^2 - \mathtt{a}^4 + \mathtt{a}^2 * \mathtt{Y}^2, \mathtt{c}^2 = \mathtt{a}^2 - \mathtt{b}^2))$$

Output :

$$-\mathtt{a}^2 * \mathtt{b}^2 + \mathtt{a}^2 * \mathtt{Y}^2 + \mathtt{b}^2 * \mathtt{X}^2$$

that is to say, after division by $a^2 * b^2$, $M$ verifies the equation :

$$\frac{X^2}{a^2} + \frac{Y^2}{b^2} = 1$$

**Another example using resultant**

Let $F1$ and $F2$ be fixed points and $A$ a variable point on the circle of 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 of center $F1$ and $x$-axis having the same direction than the vector $F1F2$, the coordinates of $A$ are :

$$A = (2a\cos(\theta); 2a\sin(\theta))$$

where $\theta$ 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 if $I$ is the middle point of $AF2$:

$$F2 = (2c, 0), \quad I = (c + ax/2; ay/2) = (c + a\frac{1-t^2}{1+t^2}; a\frac{2t1-t^2}{1+t^2})$$

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 $eq2 := diff(eq1, t) = 0$. 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 :
$(-(64\cdot \mathtt{a\^{}2}))\cdot(\mathtt{x\^{}2}+\mathtt{y\^{}2})\cdot(\mathtt{x\^{}2}\cdot\mathtt{a\^{}2}-\mathtt{x\^{}2}\cdot\mathtt{c\^{}2}+-2\cdot\mathtt{x}\cdot\mathtt{a\^{}2}\cdot\mathtt{c}+$
$2\cdot\mathtt{x}\cdot\mathtt{c\^{}3}-\mathtt{a\^{}4}+2\cdot\mathtt{a\^{}2}\cdot\mathtt{c\^{}2}+\mathtt{a\^{}2}\cdot\mathtt{y\^{}2}-\mathtt{c\^{}4})$
The factor $-64\cdot(\mathtt{x}^2+\mathtt{y}^2)$ is always different from zero, therefore the locus equation is :

$$\mathtt{x}^2\mathtt{a}^2-\mathtt{x}^2\mathtt{c}^2+-2\mathtt{x}\mathtt{a}^2\mathtt{c}+2\mathtt{x}\mathtt{c}^3-\mathtt{a}^4+2\mathtt{a}^2\mathtt{c}^2+\mathtt{a}^2\mathtt{y}^2-\mathtt{c}^4=0$$

If $O$, the middle point of $F1F2$, is choosen as origin, we find again the cartesian equation of the ellipse :

$$\frac{X^2}{a^2}+\frac{Y^2}{b^2}=1$$

## 2.26 Orthogonal polynomials

### 2.26.1 Legendre polynomials: `legendre`

`legendre` takes as argument an integer $n$ and optionnally a variable name (by default $x$).
`legendre` returns the Legendre polynomial of degree $n$ : it is a polynomial $L(n,x)$, solution of the differential equation:

$$(x^2-1).y''-2.x.y'-n(n+1).y=0$$

The Legendre polynomials verify the following recurrence relation:

$$L(0,x)=1,\quad L(1,x)=x,\quad L(n,x)=\frac{2n-1}{n}xL(n-1,x)-\frac{n-1}{n}L(n-2,x)$$

These polynomials are orthogonal for the scalar product :

$$<f,g>=\int_{-1}^{+1}f(x)g(x)\,dx$$

Input :

```
legendre(4)
```

Output :

```
(35*x^4+-30*x^2+3)/8
```

Input :

```
legendre(4,y)
```

Output :

```
(35*y^4+-30*y^2+3)/8
```

### 2.26.2 Hermite polynomial : `hermite`

`hermite` takes as argument an integer $n$ and optionnally a variable name (by default $x$).
`hermite` returns the Hermite polynomial of degree $n$.
If $H(n, x)$ denotes the Hermite polynomial of degree $n$, the following recurrence relation holds:

$$H(0,x) = 1, \quad H(1,x) = 2x, \quad H(n,x) = 2xH(n-1,x)-2(n-1)H(n-2,x)$$

These polynomials are orthogonal for the scalar product:

$$< f, g >= \int_{-\infty}^{+\infty} f(x)g(x)e^{-x^2}dx$$

Input :

```
hermite(6)
```

Output :

```
64*x^6+-480*x^4+720*x^2-120
```

Input :

```
hermite(6,y)
```

Output :

```
64*y^6+-480*y^4+720*y^2-120
```

### 2.26.3 Laguerre polynomials: `laguerre`

`laguerre` takes as argument an integer $n$ and optionnally a variable name (by default $x$) and a parameter name (by default $a$).
`laguerre` returns the Laguerre polynomial of degree $n$ and of parameter $a$.
If $L(n, a, x)$ denotes the Laguerre polynomial of degree $n$ and parameter $a$, the following recurrence relation holds:

$$L(0,a,x) = 1, \quad L(1,a,x) = 1+a-x, \quad L(n,a,x) = \frac{2n+a-1-x}{n}L(n-1,a,x)-\frac{n+a-1}{n}L(n-$$

These polynomials are orthogonal for the scalar product

$$< f, g >= \int_{0}^{+\infty} f(x)g(x)x^a e^{-x}dx$$

Input :

```
laguerre(2)
```

Output :

```
(a^2+-2*a*x+3*a+x^2+-4*x+2)/2
```

Input :

```
laguerre(2,y)
```

Output :

```
(a^2+-2*a*y+3*a+y^2+-4*y+2)/2
```

Input :

```
laguerre(2,y,b)
```

Output :

```
(b^2+-2*b*y+3*b+y^2+-4*y+2)/2
```

### 2.26.4 Tchebychev polynomials of first kind: `tchebyshev1`

`tchebyshev1` takes as argument an integer $n$ and optionnally a variable name (by default $x$).
`tchebyshev1` returns the Tchebychev polynomial of first kind of degree $n$.
The Tchebychev polynomial of first kind $T(n,x)$ is defined by

$$T(n,x)=\cos(n.\arccos(x))$$

and verify 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

$$<f,g>=\int_{-1}^{+1}\frac{f(x)g(x)}{\sqrt{1-x^2}}dx$$

Input :

```
tchebyshev1(4)
```

Output :

```
8*x^4+-8*x^2+1
```

Input :

```
tchebyshev1(4,y)
```

Output :

```
8*y^4+-8*y^2+1
```

Indeed

$$\begin{aligned}\cos(4.x) &= Re((\cos(x)+i.\sin(x))^4)\\ &= \cos(x)^4-6.\cos(x)^2.(1-\cos(x)^2)+((1-\cos(x)^2)^2\\ &= T(4,\cos(x))\end{aligned}$$

### 2.26.5 Tchebychev polynomial of second kind: `tchebyshev2`

`tchebyshev2` takes as argument an integer $n$ and optionnally a variable name (by default $x$).
`tchebyshev2` returns the Tchebychev polynomial of second kind of degree $n$.
The Tchebychev polynomial of second kind $U(n,x)$ is defined by:

$$U(n,x) = \frac{\sin((n+1).\arccos(x))}{\sin(\arccos(x))}$$

or equivalently:

$$\sin((n+1)x) = \sin(x) * U(n, \cos(x))$$

The $U(n,x)$ verifies the recurrence relation:

$$U(0,x) = 1, \quad U(1,x) = 2x, \quad U(n,x) = 2xU(n-1,x) - U(n-2,x)$$

The polynomials $U(n,x)$ are orthogonal for the scalar product

$$< f, g >= \int_{-1}^{+1} f(x)g(x)\sqrt{1-x^2}dx$$

Input :

```
tchebyshev2(3)
```

Output :

```
8*x^3+-4*x
```

Input :

```
tchebyshev2(3,y)
```

Output :

```
8*y^3+-4*y
```

Indeed:

$$\sin(4.x) = \sin(x) * (8 * \cos(x)^3 - 4.\cos(x)) = \sin(x) * U(3, \cos(x))$$

## 2.27 Gröbner basis and Gröbner reduction

### 2.27.1 Gröbner basis : `gbasis`

`gbasis` takes at least two arguments

- a vector of multivariate polynomials
- a vector of variables names,

Optionnal arguments may be used to specify the ordering and algorithms. By default, the ordering is lexicographic (with respect to the list of variable names ordering) and the polynomials are written in decreasing power orders with respect to this order. For example, the output will be like ... $+ x^2y^4z^3 + x^2y^3z^4 +$ ... if the second argument is $[x, y, z]$ because $(2, 4, 3) > (2, 3, 4)$ but the output would be like ... $+ x^2y^3z^4 + x^2y^4z^3 +$ ... if the second argument is $[x, z, y]$.
`gbasis` returns a Gröbner basis of the polynomial ideal spanned by these polynomials.

**Property**
If $I$ is an ideal and if $(G_k)_{k\in K}$ is a Gröbner basis of this ideal $I$ then, if $F$ is a non zero polynomial in $I$, the greatest monomial of $F$ is divisible by the greatest monomial of one of the $G_k$. In other words, if you do an euclidian division of $F \neq 0$ by the corresponding $G_k$, take the remainder of this division, do again the same and so on, at some point you get a null remainder.

Input :

```
gbasis([2*x*y-y^2,x^2-2*x*y],[x,y])
```

Output :

```
[4*x^2+-4*y^2,2*x*y-y^2,-(3*y^3)]
```

As indicated above, `gbasis` may have more than 2 arguments :

- `plex` (lexicographic only), `tdeg` (total degree then lexicographic order), `revlex` (total degree then inverse lexicographic order), to specify an order on the monomials (`plex` is the order by default),
- `with_cocoa=true` or `with_cocoa=false`, if you want to use the `CoCoA` library to compute the Gröbner basis (recommended, requires that `CoCoA` support compiled in)
- `with_f5=true` or `with_f5=false` for using the F5 algorithm of the `CoCoA` librairy . In this case the specify order is not used (the polynomials are homogeneized).

Input :

```
gbasis([x1+x2+x3,x1*x2+x1*x3+x2*x3,x1*x2*x3-1],
        [x1,x2,x3],tdeg,with_cocoa=false)
```

Output

```
[x3^3-1,-x2^2-x2*x3-x3^2,x1+x2+x3]
```

### 2.27.2 Gröbner reduction : `greduce`

`greduce` has three arguments : a multivariate polynomial, a vector made of polynomials which is supposed to be a Gröbner basis, and a vector of variable names.
`greduce` returns the reduction of the polynomial given as first argument with respect to the Gröbner basis given as the second argument. It is 0 if and only if the polynomial belongs to the ideal.

Input :

```
greduce(x*y-1,[x^2-y^2,2*x*y-y^2,y^3],[x,y])
```

Output :

```
y^2-2
```

that is to say $xy - 1 = \frac{1}{2}(y^2 - 2) \mod I$ where $I$ is the ideal generated by the Gröbner basis $[x^2 - y^2, 2xy - y^2, y^3]$, because $y^2 - 2$ is the euclidian division rmainder of $2(xy - 1)$ by $G_2 = 2xy - y^2$.

Like `gbasis` (cf. 2.27.1), `greduce` may have more than 3 arguments to specify ordering and algorithm if they differ from the default (lexicographic ordering). Input :

```
greduce(x1^2*x3^2,[x3^3-1,-x2^2-x2*x3-x3^2,x1+x2+x3],
                [x1,x2,x3],tdeg)
```

Output

```
x2
```

### 2.27.3 Build a polynomial from it's evaluation : `genpoly`

`genpoly` takes three arguments : a polynomial $P$ with $n-1$ variables, an integer $b$ and the name of a variable `var`.
`genpoly` returns the polynomial $Q$ with $n$ variables (the $P$ variables and the variable `var` given as second argument), such that :

- `subst(Q,var=b)==P`
- the coefficients of $Q$ belongs to the interval $]-b/2\ ;\ b/2]$

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). Input :

```
genpoly(61,6,x)
```

Output :

```
2*x^2-2*x+1
```

Indeed 61 divided by 6 is 10, remains 1, then 10 divided by 6 is 2 remains -2 (instead of the usual quotient 1 and remainder 4 out of bounds),

$$61 = 2 * 6^2 - 2 * 6 + 1$$

Input :

```
genpoly(5,6,x)
```

Output :

```
x-1
```

Indeed : $5 = 6 - 1$
Input :

```
genpoly(7,6,x)
```

Output :

```
x+1
```

Indeed : $7 = 6 + 1$
Input :

```
genpoly(7*y+5,6,x)
```

Output :

```
x*y+x+y-1
```

Indeed : $x * y + x + y - 1 = y(x + 1) + (x - 1)$
Input :

```
genpoly(7*y+5*z^2,6,x)
```

Output :

```
x*y+x*z+y-z
```

Indeed : $x * y + x * z + y - z = y * (x + 1) + z * (x - 1)$

## 2.28 Rational fractions

### 2.28.1 Numerator : `getNum`

`getNum` takes as argument a rational fraction and returns the numerator of this fraction. Unlike `numer`, `getNum` does not simplify the fraction before extracting the numerator.
Input :

```
getNum((x^2-1)/(x-1))
```

Output :

```
x^2-1
```

Input :

```
getNum((x^2+2*x+1)/(x^2-1))
```

Output :

```
x^2+2*x+1
```

### 2.28.2 Numerator after simplification : `numer`

`numer` takes as argument a rational fraction and returns the numerator of an irreducible representant of this fraction (see also 2.8.3).
Input :

```
numer((x^2-1)/(x-1))
```

Output :

```
x+1
```

Input :

```
numer((x^2+2*x+1)/(x^2-1))
```

Output :

```
x+1
```

### 2.28.3 Denominator : `getDenom`

`getDenom` takes as argument a rational fraction and returns the denominator of this fraction. Unlike `denom`, `getDenom` does not simplify the fraction before extracting the denominator.
Input :

```
getDenom((x^2-1)/(x-1))
```

Output :

```
x-1
```

Input :

```
getDenom((x^2+2*x+1)/(x^2-1))
```

Output :

```
x^2-1
```

### 2.28.4 Denominator after simplification : `denom`

`denom` (or `getDenom`) takes as argument a rational fraction and returns the denominator of an irreducible representant of this fraction (see also 2.8.4).
Input :

```
denom((x^2-1)/(x-1))
```

Output :

```
1
```

Input :

```
denom((x^2+2*x+1)/(x^2-1))
```

Output :

```
x-1
```

### 2.28.5 Numerator and denominator : `f2nd fxnd`

`f2nd` (or `fxnd`) takes as argument a rational fraction and returns the liste of the numerator and the denominator of an irreducible representant of this fraction (see also 2.8.5).
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]
```

### 2.28.6 Simplify : `simp2`

`simp2` takes as argument two polynomials (or two integers see 2.8.6). These two polynomials are seen as the numerator and denominator of a rational fraction.
`simp2` returns a list of two polynomials seen as the numerator and denominator of an irreducible representant of this rational fraction.
Input :

```
simp2(x^3-1,x^2-1)
```

Output :

```
[x^2+x+1,x+1]
```

### 2.28.7 Common denominator : `comDenom`

`comDenom` takes as argument a sum of rational fractions.
`comDenom` rewrite the sum as a unique rational fraction. The denominator of this rational fraction is the common denominator of the rational fractions given as argument.
Input :

```
comDenom(x-1/(x-1)-1/(x^2-1))
```

Output :

```
(x^3+-2*x-2)/(x^2-1)
```

### 2.28.8 Integer and fractional part : `propfrac`

`propfrac` takes as argument a rational fraction.
`propfrac` rewrites this rational fraction as the sum of it's integer part and proper fractional part.
`propfrac(A(x)/B(x))` writes the fraction $\frac{A(x)}{B(x)}$ (after reduction), as :

$$Q(x)+\frac{R(x)}{B(x)} \quad \text{where } R(x)=0 \text{ or } 0\leq \text{degree}(R(x))<\text{degree}(B(x))$$

Input :

```
propfrac((5*x+3)*(x-1)/(x+2))
```

Output :

```
5*x-12+21/(x+2)
```

### 2.28.9 Partial fraction expansion : `partfrac`

`partfrac` takes as argument a rational fraction.
`partfrac` returns the partial fraction expansion of this rational fraction.
The `partfrac` command is equivalent to the `convert` command with `parfrac` (or `partfrac` or `fullparfrac`) as option (see also 2.21.23).
**Example** :
Find the partial fraction expansion of :

$$\frac{x^5-2x^3+1}{x^4-2x^3+2x^2-2x+1}$$

Input :

```
partfrac((x^5-2*x^3+1)/(x^4-2*x^3+2*x^2-2*x+1))
```

Output in real mode :

```
x+2-1/(2*(x-1))+(x-3)/(2*(x^2+1))
```

Output in complex mode:

```
x+2+(-1+2*i)/((2-2*i)*((i)*x+1))+1/(2*(-x+1))+

(-1-2*i)/((2-2*i)*(x+i))
```

## 2.29 Exact roots of a polynomial

### 2.29.1 Exact bounds for complex roots of a polynomial : `complexroot`

`complexroot` takes 2 or 4 arguments : a polynomial and a real number $\epsilon$ and optionnally two complex numbers $\alpha, \beta$.
`complexroot` returns a list of vectors.

- If `complexroot` has 2 arguments, the elements of each vector are

  - either 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.
    Let the interval be $[a_1 + ib_1, a_2 + ib_2]$ then $|a_1 - a_2| < \epsilon$, $|b_1 - b_2| < \epsilon$ and the root $a + ib$ verifies $a_1 \leq a \leq a_2$ and $b_1 \leq b \leq b_2$.
  - or the value of an exact complex root of the polynomial and the multiplicity of this root

- If `complexroot` has 4 arguments, `complexroot` returns a list of vectors as above, but only for the roots lying in the rectangle with sides parallel to the axis having $\alpha, \beta$ as opposite vertices.

To find the roots of $x^3 + 1$, input:

```
complexroot(x^3+1,0.1)
```

Output :

```
[[-1,1],[[(4-7*i)/8,(8-13*i)/16],1],[[(8+13*i)/16,(4+7*i)/8],1]]
```

Hence, for $x^3 + 1$ :

- -1 is a root of multiplicity 1,
- 1/2+i*$b$ is a root of multiplicity 1 with $-7/8 \leq b \leq -13/16$,
- 1/2+i*$c$ is a root of multiplicity 1 with $13/1 \leq c \leq 7/8$.

To find the roots of $x^3+1$ lying inside the rectangle of opposite vertices $-1, 1+2*i$, input:

```
complexroot(x^3+1,0.1,-1,1+2*i)
```

Output :

```
[[-1,1],[[(8+13*i)/16,(4+7*i)/8],1]]
```

## 2.29.2 Exact bounds for real roots of a polynomial : `realroot`

`realroot` has 2 or 4 arguments : a polynomial and a real number $\epsilon$ and optionnally two reals numbers $\alpha, \beta$.
`realroot` returns a list of vectors.

- If `realroot` has 2 arguments, the elements of each vector are
  - either a real interval containing a real root of the polynomial and the multiplicity of this root. Let the interval be $[a_1, a_2]$ then $|a_1 - a_2| < \epsilon$ and the root $a$ verifies $a_1 \leq a \leq a_2$.
  - or the value of an exact real root of the polynomial and the multiplicity of this roots

- If `realroot` has 4 arguments, `realroot` returns a list of vectors as above, but only for the roots inside the interval $[\alpha, \beta]$.

To find the real roots of $x^3 + 1$, input:

```
realroot(x^3+1, 0.1)
```

Output :

```
[[-1,1]]
```

To find the real roots of $x^3 - x^2 - 2x + 2$, input:

```
realroot(x^3-x^2-2*x+2, 0.1)
```

Output :

```
[[1,1],[[(-3)/2,(-45)/32],1],[[45/32,3/2],1]]
```

To 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],[[11/8,23/16],1]]
```

### 2.29.3 Exact values of rational roots of a polynomial : `rationalroot`

`rationalroot` takes 1 or 3 arguments : a polynomial and optionnally two reals numbers $\alpha, \beta$.

- If `rationalroot` has 1 argument, `rationalroot` returns the list of the value of the rational roots of the polynomial without multiplicity.
- If `rationalroot` has 3 arguments, `rationalroot` returns only the rational roots of the polynomial which are in the interval $[\alpha, \beta]$.

To 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 :

```
[2,3/2,-2]
```

To 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 :

```
[2,3/2]
```

To 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 :

```
[3/2]
```

To 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 :

```
[3/2]
```

To find the rational roots of $(3*x-2)^2*(2x+1) = 18*x^3-15*x^2-4*x+4$, input:

```
rationalroot(18*x^3-15*x^2-4*x+4)
```

Output :

```
[(-1)/2,2/3]
```

### 2.29.4 Exact values of the complex rational roots of a polynomial : `crationalroot`

`crationalroot` takes 1 or 3 arguments : a polynomial and optionnally two complex numbers $\alpha, \beta$.

- If `crationalroot` has 1 argument, `crationalroot` returns the list of the complex rational roots of the polynomial without multiplicity.
- if `crationalroot` has 3 arguments, `crationalroot` returns only the complex rational roots of the polynomial which are in the rectangle with sides parallel to the axis having $[\alpha, \beta]$ as opposite vertices.

To find the rational complex roots of $(x^2+4)*(2x-3) = 2*x^3-3*x^2+8*x-12$, input :

```
crationalroot(2*x^3-3*x^2+8*x-12)
```

Output :

```
[2*i,3/2,-2*i]
```

## 2.30 Exact roots and poles

### 2.30.1 Roots and poles of a rational function : `froot`

`froot` takes a rational function $F(x)$ as argument.
`froot` returns a vector whose components are the roots and the poles of $F[x]$, each one followed by it's multiplicity.
If `Xcas` can not find the exact values of the roots or poles, it tries to find approximate values if $F(x)$ has numeric coefficients.
Input :

```
froot((x^5-2*x^4+x^3)/(x-2))
```

Output :

```
[1,2,0,3,2,-1]
```

Hence, for $F(x) = \dfrac{x^5 - 2.x^4 + x^3}{x - 2}$ :

- 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,(1+sqrt(5))/2,1,(1-sqrt(5))/2,1,2,-1]
```

**Remark** : to have the complex roots and the poles, check `Complex` in the `cas` configuration (red button giving the state line).
Input :

```
froot((x^2+1)/(x-2))
```

Output :

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

### 2.30.2 Rational function given by roots and poles : `fcoeff`

`fcoeff` has as argument a vector whose components are the roots and poles of a rational function $F[x]$, each one followed by it's multiplicity.
`fcoeff` returns the rational function $F(x)$.
Input :

```
fcoeff([1,2,0,3,2,-1])
```

Output :

```
(x-1)^2*x^3/(x-2)
```

## 2.31 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 $n\%p$. Some examples of input for
  - 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` and `o%0`,then output is `-3`.

- 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 (see also the next section).

**Remark**

- For some commands in $\mathbb{Z}/p\mathbb{Z}$ or in $\mathbb{Z}/p\mathbb{Z}[x]$, `p` must be a prime integer.
- The representation is the symetric representation :
  `11%13` returns `-2%13`.

### 2.31.1 Expand and reduce : `normal`

`normal` takes as argument a polynomial expression.
`normal` expand and reduce this expression in $\mathbb{Z}/p\mathbb{Z}[x]$.
Input :

```
normal(((2*x^2+12)*( 5*x-4))%13)
```

Output :

```
(-3%13)*x^3+(5%13)*x^2+(-5%13)*x+4%13
```

### 2.31.2 Addition in $\mathbb{Z}/p\mathbb{Z}$ or in $\mathbb{Z}/p\mathbb{Z}[x]$ : `+`

`+` adds two integers in $\mathbb{Z}/p\mathbb{Z}$, or two polynomials in $\mathbb{Z}/p\mathbb{Z}[x]$. For polynomials expressions, use the `normal` command to simplify.
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
```

### 2.31.3 Substraction in $\mathbb{Z}/p\mathbb{Z}$ or in $\mathbb{Z}/p\mathbb{Z}[x]$ : -

- substracts two integers in $\mathbb{Z}/p\mathbb{Z}$ or two polynomials in $\mathbb{Z}/p\mathbb{Z}[x]$. For polynomial expressions, use the `normal` command to simplify.
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 also :

```
normal(11% 13*x+5%13-8% 13*x+6%13)
```

Output :

```
(3%13)*x+-1%13
```

### 2.31.4 Multiplication in $\mathbb{Z}/p\mathbb{Z}$ or in $\mathbb{Z}/p\mathbb{Z}[x]$ : *

* multiplies two integers in $\mathbb{Z}/p\mathbb{Z}$ or two polynomials in $\mathbb{Z}/p\mathbb{Z}[x]$. For polynomial expressions, use the `normal` command to simplify.
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 also :

```
normal((11% 13*x+5%13)*(8% 13*x+6%13))
```

Output :

```
(-3%13)*x^2+(2%13)*x+4%13
```

### 2.31.5 Euclidian quotient : `quo`

`quo` takes as arguments two polynomials $A$ and $B$ with coefficients in $\mathbb{Z}/p\mathbb{Z}$, where $A$ and $B$ are list polynomials or symbolic polynomials with respect to $x$ or to an optionnal third argument.
`quo` returns the quotient of the euclidian division of $A$ by $B$ in $\mathbb{Z}/p\mathbb{Z}[x]$.
Input :

```
quo((x^3+x^2+1)%13,(2*x^2+4)%13)
```

Or :

```
quo((x^3+x^2+1,2*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 $-3*4 = -6*2 = 1 \mod 13$.

### 2.31.6 Euclidian remainder : `rem`

`rem` takes as arguments two polynomials $A$ and $B$ with coefficients in $\mathbb{Z}/p\mathbb{Z}$, where $A$ and $B$ are list polynomials or symbolic polynomials with respect to $x$ or to an optionnal third argument.
`rem` returns the remainder of the euclidian division of $A$ by $B$ in $\mathbb{Z}/p\mathbb{Z}[x]$.
Input :

```
rem((x^3+x^2+1)%13,(2*x^2+4)%13)
```

Or :

```
rem((x^3+x^2+1,2*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 $-3*4 = -6*2 = 1 \mod 13$.

### 2.31.7 Euclidian quotient and euclidian remainder : `quorem`

`quorem` takes as arguments two polynomials $A$ and $B$ with coefficients in $\mathbb{Z}/p\mathbb{Z}$, where $A$ and $B$ are list polynomials or symbolic polynomials with respect to $x$ or to an optionnal third argument.
`quorem` returns the list of the quotient and remainder of the euclidian division of $A$ by $B$ in $\mathbb{Z}/p\mathbb{Z}[x]$ (see also 2.6.12 and 2.25.6).
Input :

```
quorem((x^3+x^2+1)%13,(2*x^2+4)%13)
```

Or :

```
quorem((x^3+x^2+1,2*x^2+4)%13)
```

Output:

```
[(-6%13)*x+-6%13,(-2%13)*x+-1%13]
```

Indeed $x^3+x^2+1=(2x^2+4)(\frac{x+1}{2})+\frac{5x-4}{4}$
and $-3*4=-6*2=1 \mod 13$.

### 2.31.8 Division in $\mathbb{Z}/p\mathbb{Z}$ or in $\mathbb{Z}/p\mathbb{Z}[x]$ : /

/ divides two integers in $\mathbb{Z}/p\mathbb{Z}$ or two polynomials $A$ and $B$ in $\mathbb{Z}/p\mathbb{Z}[x]$.
For polynomials, the result is the irreducible representant of the fraction $\frac{A}{B}$ in $\mathbb{Z}/p\mathbb{Z}[x]$.
For integers in $\mathbb{Z}/p\mathbb{Z}$, input :

```
5%13/2% 13
```

Since 2 is invertible in $Z/13\mathbb{Z}$, we get the output :

```
-4%13
```

For polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$, input :

```
(2*x^2+5)%13/(5*x^2+2*x-3)%13
```

Output :

```
((6%13)*x+1%13)/((2%13)*x+2%13)
```

### 2.31.9 Power in $\mathbb{Z}/p\mathbb{Z}$ and in $\mathbb{Z}/p\mathbb{Z}[x]$ : ^

To compute `a` to the power `n` in $\mathbb{Z}/p\mathbb{Z}$, we use the operator ^. `Xcas` implementation is the binary power algorithm.
Input :

```
(5%13)^2
```

Output :

```
-1%13
```

To compute `A` to the power `n` in $\mathbb{Z}/p\mathbb{Z}[x]$, we use the operator ^ and the `normal` command .
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 $10=-3 \pmod{13}$ $40=1 \pmod{13}$ $80=2 \pmod{13}$ $32=6 \pmod{13}$.

### 2.31.10 Compute $a^n \mod p$ : `powmod powermod`

`powmod` (or `powermod` takes as argument $a, n, p$.
`powmod` (or `powermod`) returns $a^n \mod p$ in $[0; p-1]$.
Input :

```
powmod(5,2,13)
```

Output :

```
12
```

Input :

```
powmod(5,2,12)
```

Output :

```
1
```

### 2.31.11 Inverse in $\mathbb{Z}/p\mathbb{Z}$ : `inv inverse` or `/`

To compute the inverse of an integer `n` in $\mathbb{Z}/p\mathbb{Z}$, input `1/n%p` or `inv(n%p)` or `inverse(n%p)`.
Input :

```
inv(3%13)
```

Output :

```
-4%13
```

Indeed $3 \times -4 = -12 = 1 \pmod{13}$.

### 2.31.12 Rebuild a fraction from it's value modulo $p$ : `fracmod`

`fracmod` takes two arguments, an integer $n$ (representing a fraction) and an integer $p$ (the modulus).
If possible, `fracmod` returns a fraction $a/b$ such that

$$-\frac{\sqrt{p}}{2} < a \leq \frac{\sqrt{p}}{2}, \quad 0 \leq b < \frac{\sqrt{p}}{2}, \quad n \times b = a \pmod{p}$$

In other words $n = a/b \pmod{p}$.
Input :

```
fracmod(3,13)
```

Output :

```
-1/4
```

Indeed : $3 * -4 = -12 = 1 \pmod{13}$, hence $3 = -1/4\%13$.
Input :

```
fracmod(13,121)
```

Output :

```
-4/9
```

Indeed : $13 \times -9 = -117 = 4 \pmod{121}$ hence $13 = -4/9\%13$.

### 2.31.13 GCD in $\mathbb{Z}/p\mathbb{Z}[x]$ : `gcd`

`gcd` takes as arguments two polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$ ($p$ must be prime).
`gcd` returns the GCD of these polynomials computed in $\mathbb{Z}/p\mathbb{Z}[x]$ (see also 2.25.7 for polynomials with non modular coefficients).
Input :

```
gcd((2*x^2+5)%13,(5*x^2+2*x-3)%13)
```

Output :

```
(-4%13)*x+5%13
```

Input :

```
gcd((x^2+2*x+1,x^2-1)) mod 5)
```

Output :

```
x%5
```

Note the difference with a gcd computation in $\mathbb{Z}[X]$ followed by a reduction modulo 5, input:

```
gcd(x^2+2*x+1,x^2-1) mod 5
```

Output :

```
1
```

### 2.31.14 Factorization over $\mathbb{Z}/p\mathbb{Z}[x]$ : `factor factoriser`

`factor` takes as argument a polynomial with coefficients in $\mathbb{Z}/p\mathbb{Z}[x]$.
`factor` factorizes this polynomial in $\mathbb{Z}/p\mathbb{Z}[x]$ ($p$ must be prime).
Input :

```
factor((-3*x^3+5*x^2-5*x+4)%13)
```

Output :

```
((1%13)*x+-6%13)*((-3%13)*x^2+-5%13)
```

### 2.31.15 Determinant of a matrix in $\mathbb{Z}/p\mathbb{Z}$ : `det`

`det` takes as argument a matrix $A$ with coefficients in $Z/pZ$.
`det` returns the determinant of this matrix $A$.
Computation are done in $\mathbb{Z}/p\mathbb{Z}$ by Gauss reduction. 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
```

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`).

### 2.31.16 Inverse of a matrix with coefficients in $\mathbb{Z}/p\mathbb{Z}$ : `inv inverse`

`inverse` (or `inv`) takes as argument a matrix $A$ in $\mathbb{Z}/p\mathbb{Z}$.
`inverse` (ou `inv`) returns the inverse of the matrix $A$ in $Z/p\mathbb{Z}$.
Input :

```
inverse([[1,2,9]%13,[3,10,0]%13,[3,11,1]%13])
```

Or :

```
inv([[1,2,9]%13,[3,10,0]%13,[3,11,1]%13])
```

Or :

```
inverse([[1,2,9],[3,10,0],[3,11,1]]%13)
```

Or :

```
inv([[1,2,9],[3,10,0],[3,11,1]]%13)
```

Output :

```
[[2%13,-4%13,-5%13],[2%13,0%13,-5%13],
        [-2%13,-1%13,6%13]]
```

it is the inverse of $A = [[1, 2, 9], [3, 10, 0], [3, 11, 1]]$ in $\mathbb{Z}/13\mathbb{Z}$.

### 2.31.17 Row reduction to echelon form in $\mathbb{Z}/p\mathbb{Z}$ : `rref`

`rref` find the row reduction to echelon form of a matrix with coefficients in $\mathbb{Z}/p\mathbb{Z}$.
This may be used to solve a linear system of equations with coefficients in $\mathbb{Z}/p\mathbb{Z}$, by rewriting it in matrix form (see also 2.52.3) :

```
A*X=B
```

`rref` takes as argument the augmented matrix of the system (the matrix obtained by augmenting matrix `A` to the right with the column vector `B`).
`rref` returns a matrix `[A1,B1]` : `A1` has 1 on it's principal diagonal, and zeros outside, and the solutions in $\mathbb{Z}/p\mathbb{Z}$, of :

```
A1*X=B1
```

are the same as the solutions of:

```
A*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 :

```
[[1%13,0%13,3%13],[0%13,1%13,3%13]]
```

hence `x=3%13` and `y=3%13`.

### 2.31.18 Construction of a Galois field : `GF`

`GF` takes as arguments a prime integer $p$ and an integer $n > 1$.
`GF` returns a Galois field of caracteristic $p$ having $p^n$ elements.
Elements of the field and the field itself are represented by `GF(...)` where `...` is the following sequence:

- the caracteristic $p$ ($px = 0$),
- an irreducible primitive minimal 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 $I$,
- the name of the polynomial variable, by default `x`,
- a polynomial (a remainder modulo the minimal polynomial) for an element of the field (field elements are represented with the additive representation) or `undef` for the field itself.

You should give a name to this field (for example `G:=GF(p,n)`), in order to build elements of the field from a polynomial in $\mathbb{Z}/p\mathbb{Z}[X]$, for example `G(x^3+x)`. Note that `G(x)` is a generator of the multiplicative group $G^*$.
Input :

```
G:=GF(2,8)
```

Output :

```
GF(2,x^8-x^6-x^4-x^3-x^2-x-1,x,undef)
```

The field $G$ has $2^8 = 256$ elements and $x$ generates the multiplicative group of this field ($\{1, x, x^2, ...x^{254}\}$).
Input :

```
G(x^9)
```

Output :

```
GF(2,x^8-x^6-x^4-x^3-x^2-x-1,x,x^7+x^5+x^4+x^3+x^2+x)
```

indeed $x^8 = x^6 + x^4 + x^3 + x^2 + x + 1$, hence $x^9 = x^7 + x^5 + x^4 + x^3 + x^2 + x$.
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 on these examples, the output contains many times the same informations that you would prefer no to see if you work many times with the same field. For this reason, the definition of a Galois field may have an optionnal 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 irreductible primitive polynomial that we whish 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 :

```
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'],undef)
```

### 2.31.19 Factorize a polynomial with coefficients in a Galois field : `factor`

`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
```

## 2.32 Compute in $\mathbb{Z}/p\mathbb{Z}[x]$ using Maple syntax

### 2.32.1 Euclidean quotient : `Quo`

`Quo` is the inert form of `quo`.
`Quo` returns the euclidean quotient between two polynomials without evaluation. It is used in conjonction with `mod` in Maple syntax mode to compute the euclidean quotient of the division of two polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$.
Input in `Xcas` mode:

```
Quo((x^3+x^2+1) mod 13,(2*x^2+4) mod 13)
```

Output :

```
quo((x^3+x^2+1)%13,(2*x^2+4)%13)
```

you need to `eval(ans())` to get :

```
(-6%13)*x+-6%13
```

Input in `Maple` mode :

```
Quo(x^3+x^2+1,2*x^2+4) mod 13
```

Output :

```
(-6)*x-6
```

Input in `Maple` mode :

```
Quo(x^2+2*x,x^2+6*x+5) mod 5
```

Output :

```
1
```

### 2.32.2 Euclidien remainder: `Rem`

`Rem` is the inert form of `rem`.
`Rem` returns the euclidean remainder between two polynomials without evaluation. It is used in conjonction with `mod` in Maple syntax mode to compute the euclidean remainder of the division of two polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$.
Input in `Xcas` mode :

```
Rem((x^3+x^2+1) mod 13,(2*x^2+4) mod 13)
```

Output :

```
rem((x^3+x^2+1)%13,(2*x^2+4)%13)
```

you need to `eval(ans())` to get :

```
(-2%13)*x+-1%13
```

Input in `Maple` mode :

```
Rem(x^3+x^2+1,2*x^2+4) mod 13
```

Output :

```
(-2)*x-1
```

Input in `Maple` mode :

```
Rem(x^2+2*x,x^2+6*x+5) mod 5
```

Output :

```
1*x
```

### 2.32.3 GCD in $\mathbb{Z}/p\mathbb{Z}[x]$ : `Gcd`

`Gcd` is the inert form of `gcd`.
`Gcd` returns the gcd (greatest common divisor) of two polynomials (or of a list of polynomials or of a sequence of polynomials) without evaluation.
It is used in conjonction with `mod` in Maple syntax mode to compute the gcd of two polynomials with coefficients in $\mathbb{Z}/p\mathbb{Z}$ with $p$ prime (see also 2.25.7).
Input in `Xcas` mode :

```
Gcd((2*x^2+5,5*x^2+2*x-3)%13)
```

Output :

```
gcd((2*x^2+5)%13,(5*x^2+2*x-3)%13)
```

you need to `eval(ans())` to get :

```
(1%13)*x+2%13
```

Input in `Maple` mode :

```
Gcd(2*x^2+5,5*x^2+2*x-3) mod 13
```

Output :

```
1*x+2
```

Input:

```
Gcd(x^2+2*x,x^2+6*x+5) mod 5
```

Output :

```
1*x
```

### 2.32.4 Factorization in $\mathbb{Z}/p\mathbb{Z}[x]$ : `Factor`

`Factor` is the inert form of `factor`.
`Factor` takes as argument a polynomial.
`Factor` returns `factor` without evaluation. It is used in conjonction with `mod` in Maple syntax mode to factorize a polynomial with coefficients in $\mathbb{Z}/p\mathbb{Z}$ where $p$ must be prime.
Input in `Xcas` mode :

```
Factor((-3*x^3+5*x^2-5*x+4)%13)
```

Output :

```
factor((-3*x^3+5*x^2-5*x+4)%13)
```

you need to `eval(ans())` to get :

```
((1%13)*x+-6%13)*((-3%13)*x^2+-5%13)
```

Input in `Maple` mode :

```
Factor(-3*x^3+5*x^2-5*x+4) mod 13
```

Output :

```
-3*(1*x-6)*(1*x^2+6)
```

### 2.32.5 Determinant of a matrix with coefficients in $\mathbb{Z}/p\mathbb{Z}$ : `Det`

`Det` is the inert form of `det`.
`Det` takes as argument a matrix with coefficients in $\mathbb{Z}/p\mathbb{Z}$.
`Det` returns `det` without evaluation. It is used in conjonction with `mod` in Maple syntax mode to find the determinant of a matrix with coefficients in $\mathbb{Z}/p\mathbb{Z}$.
Input in `Xcas` mode :

```
Det([[1,2,9] mod 13,[3,10,0] mod 13,[3,11,1] mod 13])
```

Output :

```
det([[1%13,2%13,-4%13],[3%13,-3%13,0%13],
        [3%13,-2%13,1%13]])
```

you need to `eval(ans())` to get :

```
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

### 2.32.6 Inverse of a matrix in $\mathbb{Z}/p\mathbb{Z}$ : `Inverse`

`Inverse` is the inert form of `inverse`.
`Inverse` takes as argument a matrix with coefficients in $\mathbb{Z}/p\mathbb{Z}$.
`Inverse` returns `inverse` without evaluation. It is used in conjonction with `mod` in Maple syntax mode to find the inverse of a matrix with coefficients in $\mathbb{Z}/p\mathbb{Z}$.
Input in `Xcas` mode :

```
Inverse([[1,2,9] mod 13,[3,10,0] mod 13,[3,11,1]
                         mod13])
```

Output :

```
inverse([[1%13,2%13,9%13],[3%13,10%13,0%13],
               [3%13,11%13,1%13]])
```

you need to `eval(ans())` to get :

```
[[2%13,-4%13,-5%13],[2%13,0%13,-5%13],
          [-2%13,-1%13,6%13]]
```

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 :

```
[[2,-4,-5],[2,0,-5],[-2,-1,6]]
```

### 2.32.7 Row reduction to echelon form in $\mathbb{Z}/p\mathbb{Z}$ : `Rref`

`Rref` is the inert form of `rref`.
`Rref` returns `rref` without evaluation. It is used in conjonction with `mod` in Maple syntax mode to find the row reduction to echelon form of a matrix with coefficients in $\mathbb{Z}/p\mathbb{Z}$ (see also 2.52.3).
Example, solve in $\mathbb{Z}/13\mathbb{Z}$

$$\left\{\begin{array}{rcl} x + 2\cdot y & = & 9 \\ 3\cdot x + 10\cdot y & = & 0 \end{array}\right.$$

Input in `Xcas` mode :

```
Rref([[1,2,9] mod 13,[3,10,0] mod 13])
```

Output :

```
rref([[1%13, 2%13, 9%13],[3%13,10%13,0%13]])
```

you need to `eval(ans())` to get :

```
[[1%13,0%13,3%13],[0%13,1%13,3%13]]
```

and conclude that `x=3%13` and `y=3%13`.
Input in `Maple` mode :

```
Rref([[1,2,9],[3,10,0],[3,11,1]]) mod 13
```

Output :

```
[[1,0,0],[0,1,0],[0,0,1]]
```

## 2.33 Taylor and asymptotic expansions

### 2.33.1 Division by increasing power order : `divpc`

`divpc` takes three arguments : two polynomials expressions $A$, $B$ depending on $x$, such that the constant term of $B$ is not 0, and an integer $n$.
`divpc` returns the quotient $Q$ of the division of $A$ by $B$ by increasing power order, with $\texttt{degree}(Q) \leq n$ or $Q = 0$. In other words, $Q$ is the Taylor expansion at order $n$ of $\frac{A}{B}$ in the vicinity of $x = 0$.
Input :

```
divpc(1+x^2+x^3,1+x^2,5)
```

Output :

```
-x^5+x^3+1
```

Note that this command does not work on polynomials written as list of coefficients.

### 2.33.2 Taylor expansion : `taylor`

`taylor` takes from one to four arguments :

- an expression dependending of a variable (by default `x`),
- an equality variable=value (e.g. $x = a$) where to compute the Taylor expansion, by default `x=0`,
- an integer $n$, the order of the series expansion, by default `5`
- a direction `-1, 1` (for unidirectional series expansion) or `0` (for bidirectional series expansion) (by default `0`).

Note that the syntax `...,x,`$n$`,`$a$`,...` (instead of `...,x=`$a$`,`$n$`,...`) is also accepted.
`taylor` returns a polynomial in `x-a`, plus a remainder of the form:
`(x-a)^n*order_size(x-a)`
where `order_size` is a function such that,

$$\forall r > 0, \quad \lim_{x \to 0} x^r \text{order\_size}(x) = 0$$

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 $\ln(x)$.
Input :

```
taylor(sin(x),x=1,2)
```

Or (be carefull with the order of the arguments !) :

```
taylor(sin(x),x,2,1)
```

Output :

```
sin(1)+cos(1)*(x-1)+(-(1/2*sin(1)))*(x-1)^2+
            (x-1)^3*order_size(x-1)
```

**Remark**
The order returned by `taylor` may be smaller than $n$ if cancellations between numerator and denominator occur, for example

$$\text{taylor}(\frac{x^3+\sin(x)^3}{x-\sin(x)})$$

Input :

```
taylor(x^3+sin(x)^3/(x-sin(x)))
```

The output is only a 2nd-order series expansion :

```
6+-27/10*x^2+x^3*order_size(x)
```

Indeed the numerator and denominator valuation is 3, hence we loose 3 orders. To get order 4, we should ask $n = 7$, input :

```
taylor(x^3+sin(x)^3/(x-sin(x)),x=0,7)
```

Output is a 4th-order series expansion :

```
6+-27/10*x^2+x^3+711/1400*x^4+x^5*order_size(x)
```

### 2.33.3 Series expansion : `series`

`series` takes from one to four arguments :

- an expression dependending of a variable (by default `x`),
- an equality variable=value (e.g. $x = a$) where to compute the series expansion, by default `x=0`,
- an integer $n$, the order of the series expansion, by default `5`
- a direction `-1, 1` (for unidirectional series expansion) or `0` (for bidirectional series expansion) (by default `0`).

Note that the syntax `...,x,`$a$`,`$n$`,...` (instead of `...,x=`$a$`,`$n$`,...`) is also accepted.
`series` returns a polynomial in `x-a`, plus a remainder of the form:

```
(x-a)^n*order_size(x-a)
```

where `order_size` is a function such that,

$$\forall r > 0, \quad \lim_{x\to 0} x^r \text{order\_size}(x) = 0$$

The order returned by `series` may be smaller than $n$ if cancellations between numerator and denominator occur.
Examples :

- series expansion in the vicinity of x=0
  Find an series expansion of $\frac{x^3+\sin(x)^3}{x-\sin(x)}$ in the vicinity of x=0.
  Input :

  ```
  series(x^3+sin(x)^3/(x-sin(x)))
  ```

  Output is only a 2nd-order series expansion :

  ```
  6+-27/10*x^2+x^3*order_size(x)
  ```

  We have lost 3 orders because the valuation of the numerator and denominator is 3. To get a 4-th order expansion, we must therefore take $n = 7$, input:

  ```
  series(x^3+sin(x)^3/(x-sin(x)),x=0,7)
  ```

  Or :

  ```
  series(x^3+sin(x)^3/(x-sin(x)),x,0,7)
  ```

  Output is a 4th-order series expansion :

  ```
  6+-27/10*x^2+x^3+711/1400*x^4+ x^5*order_size(x)
  ```

- series expansion in the vicinity of x=a
  Find a series 4th-order expansion of $\cos(2x)^2$ in the vicinity of $x = \frac{\pi}{6}$.
  Input:

  ```
  series(cos(2*x)^2,x=pi/6, 4)
  ```

  Output :

  ```
  1/4+(-(4*sqrt(3)))/4*(x-pi/6)+(4*3-4)/4*(x-pi/6)^2+
  32*sqrt(3)/3/4*(x-pi/6)^3+(-16*3+16)/3/4*(x-pi/6)^4+
              (x-pi/6)^5*order_size(x-pi/6)
  ```

- series expansion in the vicinity of x=+$\infty$ or x=-$\infty$

  1. Find a 5th-order series expansion of $\arctan(x)$ in the vicinity of x=+$\infty$.
     Input :

     ```
     series(atan(x),x=+infinity,5)
     ```

     Output :

     ```
     pi/2-1/x+1/3*(1/x)^3+1/-5*(1/x)^5+
             (1/x)^6*order_size(1/x)
     ```

     Note that the expansion variable and the argument of the `order_size` function is $h = \frac{1}{x} \rightarrow_{x\rightarrow+\infty} 0$.

2. Find a series 2nd-order expansion of $(2x-1)e^{\frac{1}{x-1}}$ in the vicinity of x=+∞.
   Input :
   ```
   series((2*x-1)*exp(1/(x-1)),x=+infinity,3)
   ```
   Output is only a 1st-order series expansion :
   ```
   2*x+1+2/x+(1/x)^2*order_size(1/x)
   ```
   To get a 2nd-order series expansion in $1/x$, input:
   ```
   series((2*x-1)*exp(1/(x-1)),x=+infinity,4)
   ```
   Output :
   ```
   2*x+1+2/x+17/6*(1/x)^2+(1/x)^3*order_size(1/x)
   ```
3. Find a 2nd-order series expansion of $(2x-1)e^{\frac{1}{x-1}}$ in the vicinity of x=-∞.
   Input :
   ```
   series((2*x-1)*exp(1/(x-1)),x=-infinity,4)
   ```
   Output :
   ```
   -2*(-x)+1-2*(-1/x)+17/6*(-1/x)^2+
        (-1/x)^3*order_size(-1/x)
   ```

- unidirectional series expansion.
  The fourth parameter indicates the direction :
  - `1` to do an series expansion in the vicinity of $x = a$ with $x > a$,
  - `-1` to do an series expansion in the vicinity of $x = a$ with $x < a$,
  - `0` to do an series expansion in the vicinity of $x = a$ with $x \neq a$.

  For example, 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)
  ```
  Output :
  ```
  exp(1)/x^3+(-(exp(1)))/2/x^2+1/x*order_size(x)
  ```

### 2.33.4 Résidu d'une expression en un point : `residue`

`residue` takes as argument an expression depending of a variable, this variable name and a complex $a$ or an expression depending of a variable and the equality : variable_name=$a$. `residue` returns the residue of this exppression at the point $a$.
Input :
```
residue(cos(x)/x^3,x,0)
```
Or :
```
residue(cos(x)/x^3,x=0)
```
Output :
```
(-1)/2
```

### 2.33.5 Padé expansion: `pade`

`pade` takes 4 arguments

- an expression,
- the variable name the expression depends on,
- an integer $n$ or a polynomial $N$,
- an integer $p$.

`pade` returns a rationnal fraction $P/Q$ such that `degree(P)`$< p$ and $P/Q = f \pmod{x^{n+1}}$ or $P/Q = f \pmod{N}$. In the first case, it means that $P/Q$ and $f$ have the same Taylor expansion at 0 up to order $n$.
Input :

```
pade(exp(x),x,5,3)
```

Or :

```
pade(exp(x),x,x^6,3)
```

Output :

```
(3*x^2+24*x+60)/(-x^3+9*x^2-36*x+60)
```

To verify input :

```
taylor((3*x^2+24*x+60)/(-x^3+9*x^2-36*x+60))
```

Output :

```
1+x+1/2*x^2+1/6*x^3+1/24*x^4+1/120*x^5+x^6*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 :

```
(-2*x^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*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*order_size(x)
```

These two expressions have the same 14th-order series expansion at $x = 0$.

## 2.34 Intervals

### 2.34.1 Definition of an interval : `a1..a2`

An interval is represented by two real numbers separated by `..`, for example

```
1..3
1.2..sqrt(2)
```

Input :

```
A:=1..4

B:=1.2..sqrt(2)
```

**Warning!**
The order of the boundaries of the interval is significant. For example, if you input

```
B:=2..3; C:=3..2,
```

then `B` and `C` are different, `B==C` returns `0`.

### 2.34.2 Boundaries of an interval : `left right`

`left` (resp `right`) takes as argument an interval.
`left` (resp `right`) returns the left (resp right) boundary of this interval.
Note that `..` is an infixed operator, therefore:

- `sommet(1..5)` is equal to `'..'` and `feuille(1..5)` is equal to `(1,5)`.
- the name of the interval followed by `[0]` returns the operator `..`

- the name of the interval followed by `[1]` (or the `left` command) returns the left boundary.
- The name of the interval followed by `[2]` (or the `right` command) returns the right boundary.

Input :

```
(3..5)[0]
```

Or :

```
sommet(3..5)
```

Output :

```
'..'
```

Input :

```
left(3..5)
```

Or :

```
(3..5)[1]
```

Or :

```
feuille(3..5)[0]
```

Or :

```
op(3..5)[0]
```

Output :

```
3
```

Input :

```
right(3..5)
```

Or :

```
(2..5)[2]
```

Or :

```
feuille(3..5)[1]
```

Or :

```
op(3..5)[1]
```

Output :

```
5
```

**Remark**
`left` (resp `right`) returns also the left (resp right) member of an equation (for example `left(2*x+1=x+2)` returns `2*x+1`).

### 2.34.3 Center of an interval : `interval2center`

`interval2center` takes as argument an interval or a list of intervals.
`interval2center` returns the center of this interval or the list of centers of these intervals.
Input :

```
interval2center(3..5)
```

Output :

```
4
```

Input :

```
interval2center([2..4,4..6,6..10])
```

Output :

```
[3,5,8]
```

### 2.34.4 Intervals defined by their center : `center2interval`

`center2interval` takes as argument a vector `V` of reals and optionnally a real as second argument (by default `V[0]-(V[1]-V[0])/2`).
`center2interval` returns a vector of intervals having the real values of the first argument as centers, where the value of the second argument is the left boundary of the first interval.
Input :

```
center2interval([3,5,8])
```

Or (since the default value is 3-(5-3)/2=2) :

```
center2interval([3,5,8],2)
```

Output :

```
[2..4,4..6,6..10]
```

Input :

```
center2interval([3,5,8],2.5)
```

Output :

```
[2.5..3.5,3.5..6.5,6.5..9.5]
```

## 2.35 Sequences

### 2.35.1 Definition : `seq[] ()`

A sequence is represented by a sequence of elements separated by commas, without delimiters or with either `( )` or `seq[...]` as delimiters, for example

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

Input :

`A:=(1,2,3,4)` or `A:=seq[1,2,3,4]`

`B:=(5,6,3,4)` or `B:=seq[5,6,3,4]`

#### Remarks

- The order of the elements of the sequence is significant. For example, if `B:=(5,6,3,4)` and `C:=(3,4,5,6)`, then `B==C` returns `0`.
- (see also 2.35.5)
  `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)`

### 2.35.2 Concat two sequences : `,`

The infix operator `,` concatenates two sequences.
Input :

```
A:=(1,2,3,4)

B:=(5,6,3,4)

A,B
```

Output :

```
(1,2,3,4,5,6,3,4)
```

### 2.35.3 Get an element of a sequence : `[]`

The elements of a sequence have indexes begining at 0 in `Xcas` mode or 1 in other modes.
A sequence or a variable name assigned to a sequence followed by `[n]` returns the element of index `n` of the sequence.
Input :

```
(0,3,2)[1]
```

Output :

```
3
```

### 2.35.4 Sub-sequence of a sequence : `[]`

A sequence or a variable name assigned to a sequence followed by `[n1..n2]` returns the sub-sequence of this sequence starting at index `n1` and ending at index `n2`.
Input :

```
(0,1,2,3,4)[1..3]
```

Output :

```
(1,2,3)
```

### 2.35.5 Make a sequence or a list : `seq $`

`seq` takes two, three, four or five arguments : the first argument is an expression depending of a parameter (for example $j$) and the remaining argument(s) describe which values of $j$ will be used to generate the sequence. More precisely $j$ is assumed to move from $a$ to $b$:

- with a default step of 1 or -1: `j=a..b` or `j,a..b` (Maple-like syntax), `j,a,b` (TI-like syntax)
- or with a specific step: `j=a..b,p` (Maple-like syntax), `j,a,b,p` (TI-like syntax).

If the Maple-like syntax is used, `seq` returns a sequence, if the TI-like syntax is used, `seq` returns a list.

`$` is the infixed version of `seq` when `seq` has only two arguments and returns always a sequence.
**Remark:**

- In `Xcas` mode, the precedence of `$` is not the same as for example in `Maple`, in case of doubt put the arguments of `$` 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)`. The equivalent of `seq(4,3)` is `4$3` and returns `(4,4,4)`.
- 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(...)`.

In summary, the different way to build a sequence are :

- with `Maple`-like **syntax**

  1. `seq` has two arguments either an expression depending of a parameter (for example $j$) and $j = a..b$ where $a$ and $b$ are reals, or a constant expression and an integer $n$.
     `seq` returns the sequence where $j$ is replaced in the expression by $a$, $a + 1$,...,$b$ if $b > a$ and by $a$, $a - 1$,...,$b$ if $b < a$, or `seq` returns the sequence made by copying $n$ times the constant.

2. `seq` has three arguments an expression depending of a parameter (for example $j$) and $j = a..b, p$ where $a$, $b$ are reals and $p$ is a real number. `seq` returns the sequence where $j$ is replaced in the expression by $a$, $a+p$,...,$b$ if $b > a$ and by $a$, $a-p$,...,$b$ if $b < a$.
   Note that $j, a..b$ is also valid but $j, a..b, p$ is not valid.

- `TI` **syntax**

  1. `seq` has four arguments an expression depending of a parameter (for example $j$), the name of the parameter (for example $j$), $a$ and $b$ where $a$ and $b$ are reals.
     `seq` returns the list where $j$ is replaced in the expression by $a$, $a+1$,...,$b$ if $b > a$ and by $a$, $a-1$,...,$b$ if $b < a$.
  2. `seq` has five arguments an expression depending of a parameter (for example $j$), the name of the parameter (for example $j$), $a$, $b$ and $p$ where $a$, $b$ and $p$ are reals.
     `seq` returns the list where $j$ is substitued in the expression by $a$, $a+p$,...,$a+k*p$ ($a+k*p \leq b < a+(k+1)*p$ or $a+k*p \geq b > a+(k+1)*p$). By default, $p$=1 if $b > a$ and $p$=-1 if $b < a$.

**Note** that in `Maple` syntax, `seq` takes no more than 3 arguments and returns a sequence as in `TI` syntax, `seq` takes at least 4 arguments and returns a list.
Input to have a sequence with same elements :

```
seq(t,4)
```

Or :

```
seq(t,k=1..4)
```

Or :

```
t$4
```

Output :

```
(t,t,t,t)
```

Input to have a sequence :

```
seq(j^3,j=1..4)
```

Or :

```
(j^3)$(j=1..4)
```

Or :

```
seq(j^3,j,1..4)
```

Output :

```
(1,4,9,16)
```

Input to have a sequence :

```
seq(j^3,j=-1..4,2)
```

Output :

```
(1,1,9)
```

Or to have a list,
Input :

```
seq(j^3,j,1,4)
```

Output :

```
[1,4,9,16]
```

Input :

```
seq(j^3,j,0,5,2)
```

Output :

```
[0,8,64]
```

Input :

```
seq(j^3,j,5,0,-2)
```

or

```
seq(j^3,j,5,0,2)
```

Output :

```
[125,27,1]
```

Input :

```
seq(j^3,j,1,3,0.5)
```

Output :

```
[1,3.375,8,15.625,27]
```

Input :

```
seq(j^3,j,1,3,1/2)
```

Output :

```
[1,27/8,8,125/8,27]
```

**Examples**

- Find the third derivative of $\ln(t)$, input:

  ```
  diff(log(t),t$3)
  ```

  Output :

```
-((-(2*t))/t^4)
```

- Input :

  ```
  l:=[[2,3],[5,1],[7,2]]
  ```

  ```
  seq((l[k][0])$(l[k][1]),k=0 .. size(l)-1)
  ```

  Output :

  ```
  2,2,2,seq[5],7,7
  ```

  then `eval(ans())` returns:

  ```
  2,2,2,5,7,7
  ```

- Input to transform a string into the list of its characters :

  ```
  f(chn):={
   local l;
   l:=size(chn);
   return seq(chn[j],j,0,l-1);
  }
  ```

  then input:

  ```
  f("abracadabra")
  ```

  Output :

  ```
  ["a","b","r","a","c","a","d","a","b","r","a"]
  ```

### 2.35.6 Transform a sequence into a list : `[]` `nop`

To transform a sequence into list, just put square brackets (`[]`) around the sequence or use the command `nop`.
Input :

```
[seq(j^3,j=1..4)]
```

or :

```
seq(j^3,j,1,4)
```

or :

```
[(j^3)$(j=1..4)]
```

Output :

```
[1,4,9,16]
```

Input :

```
nop(1,4,9,16)
```

Output :

```
[1,4,9,16]
```

### 2.35.7 The + operator applied on sequences

The infixed operator +, with two sequences as argument, returns the total sum of the elements of the two sequences.
Note the difference with the lists, where the term by term sums of the elements of the two lists would be returned.
Input :

```
(1,2,3,4,5,6)+(4,3,5)
```

Or :

```
'+'((1,2,3,4,5,6),(4,3,5))
```

Output :

```
33
```

But input :

```
[1,2,3,4,5,6]+[4,3,5]
```

Output :

```
[5,5,8,4,5,6]
```

**Warning**
When the operator + is prefixed, it has to be quoted (`'+'`).

## 2.36 Sets

### 2.36.1 Definition : `set[]`

To define a set of elements, put the elements separated by a comma, with `%{ ... %}` or `set[ ... ]` as delimiters.
Input :

```
%{1,2,3,4%}
set[1,2,3,4]
```

In the `Xcas` answers, the set delimiters are displayed as ⟦ and ⟧ in order not to confuse sets with lists. 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 distincts. If you input `B:=%{5,5,6,3,4%}` and `C:=%{3,4,5,3,6%}`, then `B==C` will return `1`.

### 2.36.2 Union of two sets or of two lists : `union`

`union` is an infixed operator.
`union` takes as argument two sets or two lists `union` returns the union set of the arguments.
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⟧
```

### 2.36.3 Intersection of two sets or of two lists : `intersect`

`intersect` is an infixed operator.
`intersect` takes as argument two sets or two lists.
`intersect` returns the intersection set of the arguments.
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⟧
```

### 2.36.4 Difference of two sets or of two lists : `minus`

`minus` is an infixed operator.
`minus` takes as argument two sets or two lists.
`minus` returns the difference set of the arguments.
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⟧
```

## 2.37 Lists and vectors

### 2.37.1 Definition

A list (or a vector) is delimited by `[ ]`, it's elements must be separated by a comma. For example `[1,2,5]` is a list of three integers.

Lists can contain lists (for example, a matrice is a list of lists of the same size). Lists may be used to represent vectors (list of coordinates), matrices, univariate polynomials (list of coefficients by decreasing order).

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 repeted in a list (unlike in a set where each element is unique).

In `Xcas` answers :

- vector (or lists) delimiters are displayed as `[]`,
- matrix delimiters are displayed as **[]**,
- polynomial delimiters are displayed as [] [],
- set delimiters are displayed as ⟦ ⟧.

The list elements are indexed starting from 0 in Xcas syntax mode and from 1 in all other syntax mode.

### 2.37.2 Get an element or a sub-list of a list : `at` `[]`

#### Get an element

The $n$-th element of a list `l` of size $s$ is addressed by `l[n]` where $n$ is in $[0..s-1]$ or $[1..s]$. The equivalent prefixed function is `at`, which takes as argument a list and an integer `n`.
`at` returns the element of the list of index `n`.
Input :

```
[0,1,2][1]
```

or

```
at([0,1,2],1)
```

Output :

```
1
```

#### Extract a sub-list

If $l$ is a list of size $s$, `l[n1..n2]` returns the list extracted from `l` containing the elements of indices $n_1$ 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. The equivalent prefixed function is `at` with a list and an interval of integers (`n1..n2`) as arguments.
**See also** : `mid`, section 2.37.3.
Input :

```
[0,1,2,3,4][1..3]
```

or

```
at([0,1,2,3,4],1..3)
```

Output :

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

**Warning**
`at` can not be used for sequences, index notation must be used, like in `(0,1,2,3,4,5)[2..3]`.

### 2.37.3 Extract a sub-list : `mid`

**See also :** `at` section 2.37.2.
`mid` is used to extract a sub-list of a list.
`mid` takes as argument a list, the index of the begining of the sub-list and the length of the sub-list.
`mid` returns the sub-list.
Input :

```
mid([0,1,2,3,4,5],2,3)
```

Output :

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

**Warning**
`mid` can not be used to extract a subsequence of a sequence, because the arguments of `mid` would be merged with the sequence. Index notation must be used, like e.g. `(0,1,2,3,4,5)[2..3]`.

### 2.37.4 Get the first element of a list : `head`

`head` takes as argument a list.
`head` returns the first element of this list.
Input :

```
head([0,1,2,3])
```

Output :

```
0
```

`a:=head([0,1,2,3])` does the same thing than `a:=[0,1,2,3][0]`

### 2.37.5 Remove an element in a list : `suppress`

`suppress` takes as argument a list and an integer `n`.
`suppress` returns the list where the element of index `n` is removed.
Input :

```
suppress([3,4,2],1)
```

Output :

```
[3,2]
```

### 2.37.6 Remove the first element : `tail`

`tail` takes as argument a list. `tail` returns the list without it's first element.
Input :

```
tail([0,1,2,3])
```

Output :

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

`l:=tail([0,1,2,3])` does the same thing as `l:=suppress([0,1,2,3],0)`

### 2.37.7 Reverse order in a list : `revlist`

`revlist` takes as argument a list (resp sequence).
`revlist` returns the list (resp sequence) in the reverse order.
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]
```

### 2.37.8 Reverse a list starting from its n-th element : `rotate`

`rotate` takes as argument a list and an integer `n` (by default `n=-1`).
`rotate` rotates the list by `n` places to the left if `n>0` or 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 first.
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]
```

### 2.37.9 Permuted list from its n-th element : `shift`

`shift` takes as argument a list `l` and an integer `n` (by default `n`=-1).
`shift` rotates the list to the left if `n>0` or to the right if `n<0`. Elements leaving the list from one side are replaced by `undef` on the other side.
Input :

```
shift([0,1,2,3,4])
```

Output :

```
[undef,0,1,2,3]
```

Input :

```
shift([0,1,2,3,4],2)
```

Output :

```
[2,3,4,undef,undef]
```

Input :

```
shift([0,1,2,3,4],-2)
```

Output :

```
[undef,undef,0,1,2]
```

### 2.37.10 Modify an element in a list : `subsop`

`subsop` modifies an element in a list. `subsop` takes as argument a list and an equality (an index=a new value) in all syntax modes, but in `Maple` syntax mode the order of the arguments is reversed.
**Remark** If the second argument is `'k=NULL'`, the element of index `k` is removed of the list.
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]
```

In `Maple` mode the arguments are permuted and the index of the first element is 1.
Input :

```
subsop(2=5,[0,1,2])
```

Or :

```
L:=[0,1,2];L[2]:=5
```

Output :

```
[0,5,2]
```

### 2.37.11 Transform a list into a sequence : `op makesuite`

`op` or `makesuite` takes as argument a list.
`op` or `makesuite` transforms this list into a sequence.
See 2.15.3 for other usages of `op`.
Input :

```
op([0,1,2])
```

Or :

```
makesuite([0,1,2])
```

Output :

```
(0,1,2)
```

### 2.37.12 Transform a sequence into a list : `makevector []`

Square brackets put around a sequence transform this sequence into a list or vector. The equivalent prefixed function is `makevector` which takes a sequence as argument.
`makevector` transforms this sequence into a list or vector.
Input :

```
makevector(0,1,2)
```

Output :

```
[0,1,2]
```

Input :

```
a:=(0,1,2)
```

Input :

```
[a]
```

Or :

```
makevector(a)
```

Output :

```
[0,1,2]
```

### 2.37.13 Length of a list : `size nops length`

`size` or `nops` or `length` takes as argument a list (resp sequence).
`size` or `nops` or `length` returns the length of this list (resp sequence).
Input :

```
nops([3,4,2])
```

ou

```
size([3,4,2])
```

ou

```
length([3,4,2])
```

Output :

```
3
```

### 2.37.14 Sizes of a list of lists : `sizes`

`sizes` takes as argument a list of lists.
`sizes` returns the list of the lengths of these lists.
Input :

```
sizes([[3,4],[2]])
```

Output :

```
[2,1]
```

### 2.37.15 Concatenate two lists or a list and an element : `concat augment`

`concat` (or `augment`) takes as argument a list and an element or two lists.
`concat` (or `augment`) concats this list and this element, or concats these two lists.
Input :

```
concat([3,4,2],[1,2,4])
```

Or :

```
augment([3,4,2],[1,2,4])
```

Output :

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

Input :

```
concat([3,4,2],5)
```

Or :

```
augment([3,4,2],5)
```

Output :

```
[3,4,2,5]
```

**Warning** If you input :

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

or

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

the output will be:

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

### 2.37.16 Append an element at the end of a list : `append`

`append` takes as argument a list and an element.
`append` puts this element at the end of this list.
Input :

```
append([3,4,2],1)
```

Output :

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

Input :

```
append([1,2],[3,4])
```

Output :

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

### 2.37.17 Prepend an element at the begining of a list : `prepend`

`prepend` takes as argument a list and an element.
`prepend` puts this element at the begining of this list.
Input :

```
prepend([3,4,2],1)
```

Output :

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

Input :

```
prepend([1,2],[3,4])
```

Output :

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

### 2.37.18 Sort : `sort`

`sort` takes as argument a list or an expression.

- For a list,
  `sort` returns the list sorted in increasing order.
  Input :

  ```
  sort([3,4,2])
  ```

  Output :

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

- For an expression,
  `sort` sorts and collects terms in sums and products.
  Input :

  ```
  sort(exp(2*ln(x))+x*y-x+y*x+2*x)
  ```

  Output :

  ```
  2*x*y+exp(2*ln(x))+x
  ```

  Input :

  ```
  simplify(exp(2*ln(x))+x*y-x+y*x+2*x)
  ```

  Output :

  ```
  x^2+2*x*y+x
  ```

`sort` accepts an optionnal second argument, which is a bivariate function returning 0 or 1. If provided, this function will be used to sort the list, for example `(x,y)->x>=y` may be used as second argument to sort the list in decreasing order. This may also be used to sort list of lists (that `sort` with one argument does not know how to sort).
Input :

```
sort([3,4,2],(x,y)->x>=y)
```

Output :

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

### 2.37.19 Sort a list by increasing order : `SortA`

`SortA` takes as argument a list.
`SortA` returns this list sorted by increasing order.
Input :

```
SortA([3,4,2])
```

Output :

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

`SortA` may have a matrix as argument and in this case, `SortA` modifies the order of columns by sorting the first matrix row by increasing order.
Input :

```
SortA([[3,4,2],[6,4,5]])
```

Output :

```
[[2,3,4],[5,6,4]]
```

### 2.37.20 Sort a list by decreasing order : `SortD`

`SortD` takes a list as argument.
`SortD` returns this list sorted by decreasing order.
Input :

```
SortD([3,4,2])
```

Output :

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

`SortD` may have a matrix as argument and in this case, `SortD` modifies the order of columns by sorting the first matrix row by decreasing order.
Input :

```
SortD([[3,4,2],[6,4,5]])
```

Output :

```
[[4,3,2],[4,6,5]]
```

### 2.37.21 Select the elements of a list : `select`

`select` takes as arguments : a boolean function `f` and a list `L`.
`select` selects in the list `L`, the elements `c` such that `f(c)==true`.
Input :

```
select(x->(x>=2),[0,1,2,3,1,5])
```

Output :

```
[2,3,5]
```

### 2.37.22 Remove elements of a list : `remove`

`remove` takes as argument : a boolean function `f` and a list `L`.
`remove` removes in the list `L`, the elements `c` such that `f(c)==true`.
Input :

```
remove(x->(x>=2),[0,1,2,3,1,5])
```

Output :

```
[0,1,1]
```

**Remark** The same applies on strings, for example, to remove all the "a" of a string:
Input :

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

Output :

```
97
```

Input :

```
f(chn):={
  local l:=length(chn)-1;
  return remove(x->(ord(x)==97),seq(chn[k],k,0,l));
}
```

Then, 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"
```

### 2.37.23 Test if a value is in a list : `member`

`member` takes as argument a value `c` and a list (or a set) `L`.
`member` is a function that tests if `c` is an element of the list `L`.
`member` returns `0` if `c` is not in `L`, or a strictly positive integer which is 1 plus the index of the first occurence of `c` in `L`.
Note the order of the arguments (required for compatibility reasons)
Input :

```
member(2,[0,1,2,3,4,2])
```

Output :

```
3
```

Input :

```
member(2,%{0,1,2,3,4,2%})
```

Output :

```
3
```

### 2.37.24 Test if a value is in a list : `contains`

`contains` takes as argument a list (or a set) `L` and a value `c`.
`contains` tests if `c` is an element of the list `L`.
`contains` returns `0` if `c` is not in `L`, or a strictly positive integer which is 1+the index of the first occurence of `c` in `L`.
Input :

```
contains([0,1,2,3,4,2],2)
```

Output :

```
3
```

Input :

```
contains(%{0,1,2,3,4,2%},2)
```

Output :

```
3
```

### 2.37.25 Sum of list (or matrix) elements transformed by a function : `count`

`count` takes as argument : a real function `f` and a list `l` of length `n` (or a matrix `A` of dimension `p*q`).
`count` applies the function to the list (or matrix) elements and returns their sum, i.e. :
`count(f,l)` returns `f(l[0])+f(l[1])+...+f(l[n-1])` or
`count(f,A)` returns `f(A[0,0])+....+f(A[p-1,q-1])`.
If `f` is a boolean function `count` returns the number of elements of the list (or of the matrix) for which the boolean function is true.
Input :

```
count((x)->x,[2,12,45,3,7,78])
```

Output :

```
147
```

because : 2+12+45+3+7+78=147.
Input :

```
count((x)->x<12,[2,12,45,3,7,78])
```

Output :

```
3
```

Input :

```
count((x)->x==12,[2,12,45,3,7,78])
```

Output :

```
1
```

Input :

```
count((x)->x>12,[2,12,45,3,7,78])
```

Output :

```
2
```

Input :

```
count(x->x^2,[3,5,1])
```

Output :

```
35
```

Indeed $32 + 52 + 11 = 35$.
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.

### 2.37.26 Number of elements equal to a given value : `count_eq`

`count_eq` takes as argument : a real and a real list (or matrix).
`count_eq` returns the number of elements of the list (or matrix) which are equal to the first argument.
Input :

```
count_eq(12,[2,12,45,3,7,78])
```

Output :

```
1
```

### 2.37.27 Number of elements smaller than a given value : `count_inf`

`count_inf` takes as argument : a real and a real list (or matrix).
`count_inf` returns the number of elements of the list (or matrix) which are strictly inferior to the first argument.
Input :

```
count_inf(12,[2,12,45,3,7,78])
```

Output :

```
3
```

### 2.37.28 Number of elements greater than a given value : `count_sup`

`count_sup` takes as argument : a real and a real list (or matrix).
`count_sup` returns the number of elements of the list (or matrix) which are strictly greater to the first argument.
Input :

```
count_sup(12,[2,12,45,3,7,78])
```

Output :

```
2
```

### 2.37.29 Sum of elements of a list : `sum add`

`sum` or `add` takes as argument a list `l` (resp sequence) of reals.
`sum` or `add` returns the sum of the elements of `l`.
Input :

```
sum(2,3,4,5,6)
```

Output :

```
20
```

### 2.37.30 Cumulated sum of the elements of a list : `cumSum`

`cumSum` takes as argument a list `l` (resp sequence) of numbers or of strings. `cumSum` returns the list (resp sequence) with same length as `l` and with $k$-th element the sum (or concatenation) of the elements `l`[0], .., `l`[`k`].
Input :

```
cumSum(sqrt(2),3,4,5,6)
```

Output :

```
sqrt(2),3+sqrt(2),3+sqrt(2)+4,3+sqrt(2)+4+5,
3+sqrt(2)+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"
```

### 2.37.31 Product : `product mul`

See also 2.37.31, 2.42.6 and 2.42.8).

#### Product of values of an expression : `product`

`product(expr,var,a,b,p)` or `mul(expr,var,a,b,p)` returns the product of values of an expression `ex` when the variable `var` goes from `a` to `b` with a step `p` (by default p=1) : this syntax is for compatiblity with Maple.
Input :

```
product(x^2+1,x,1,4)
```

Or:

```
mul(x^2+1,x,1,4)
```

Output :

```
1700
```

Indeed $2 * 5 * 10 * 17 = 1700$
Input :

```
product(x^2+1,x,1,5,2)
```

Or:

```
mul(x^2+1,x,1,5,2)
```

Output :

```
520
```

Indeed $2 * 10 * 26 = 520$

#### Product of elements of a list : `product`

`product` or `mul` takes as argument a list `l` of reals (or floating numbers) or two lists of the same size (see also 2.37.31, 2.42.6 and 2.42.8).

- if `product` or `mul` has a list `l` as argument, `product` or `mul` returns the product of the elements of `l`.
  Input :

  ```
  product([2,3,4])
  ```

  Input :

  ```
  mul([2,3,4])
  ```

  Output :

  ```
  24
  ```

  Input :

  ```
  product([[2,3,4],[5,6,7]])
  ```

Output :

```
[10,18,28]
```

- if `product` or `mul` takes as arguments `l1` and `l2` (two lists or two matrices), `product` or `mul` returns the term by term product of the elements of `l1` and `l2`.
  Input :

  ```
  product([2,3,4],[5,6,7])
  ```

  Or :

  ```
  mul([2,3,4],[5,6,7])
  ```

  Output :

  ```
  [10,18,28]
  ```

  Input :

  ```
  product([[2,3,4],[5,6,7]],[[2,3,4],[5,6,7]])
  ```

  Or :

  ```
  mul([[2,3,4],[5,6,7]],[[2,3,4],[5,6,7]])
  ```

  Output :

  ```
  [[4,9,16],[25,36,49]]
  ```

### 2.37.32 Apply a function of one variable to the elements of a list : `map apply of`

`map` or `apply` or `of` applies a function to a list of elements.
`of` is the prefixed function equivalent to the parenthesis : `Xcas` translates `f(x)` internally to `of(f,x)`. It is more natural to call `map` or `apply` than `of`. Be carefull with the order of arguments (that is required for compatibility reasons).
Note that `apply` returns a list (`[]`) even if the second argument is not a list.
Input :

```
apply(x->x^2,[3,5,1])
```

or

```
of(x->x^2,[3,5,1])
```

or

```
map([3,5,1],x->x^2)
```

or first define the function $h(x) = x2$, input :

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

then

```
apply(h,[3,5,1])
```

or

```
of(h,[3,5,1])
```

or

```
map([3,5,1],h)
```

Output :

```
[9,25,1]
```

Next example, define the function $g(x) = [x, x2, x3]$, input :

```
g:=(x)->[x,x^2,x^3]
```

then

```
apply(g,[3,5,1])
```

or

```
of(g,[3,5,1])
```

or

```
map([3,5,1],g)
```

Output :

```
[[3,9,27],[5,25,125],[1,1,1]]
```

**Warning!!!** first purge `x` if `x` is not symbolic.
Note that if `l1,l2,l3` are lists `sizes([l1,l2,l3])` is equivalent to `map(size,[l1,l2,l3]`.

### 2.37.33 Apply a bivariate function to the elements of two lists : `zip`

`zip` applies a bivariate function to the elements of 2 lists.
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
```

then,

```
zip(f,[4,2,1],[3,5,1])
```

Output :

```
[25,29,2]
```

Input :

```
f:=(x,y)->[x^2+y^2,x+y]
```

puis,

```
zip(f,[4,2,1],[3,5,1])
```

Output :

```
[[25,7],[29,7],[2,2]]
```

### 2.37.34 Make a list with zeros : `newList`

`newList(n)` makes a list of `n` zeros.
Input :

```
newList(3)
```

Output :

```
[0,0,0]
```

### 2.37.35 Make a list with a function : `makelist`

`makelist` takes as argument a function `f`, the bounds `a,b` of an index variable and a step `s` (by default 1 or -1 depending on the bounds order).
`makelist` makes the list `[f(a),f(a+p)...f(a+k*p)]` with $k$ such that : $a < a+k*p \leq b < a+(k+1)*p$ or $a > a+k*p \geq b > a+(k+1)*p$.
Input :

```
makelist(x->x^2,3,5)
```

or

```
makelist(x->x^2,3,5,1)
```

or first define the function $h(x) = x2$ by `h(x):=x^2` then input

```
makelist(h,3,5,1)
```

Output :

```
[9,16,25]
```

Input :

```
makelist(x->x^2,3,6,2)
```

Output :

```
[9,25]
```

**Warning!!!** purge `x` if `x` is not symbolic.

### 2.37.36 Make a random vector or list : `randvector`

`randvector` takes as argument an integer $n$ and optionnally a second argument, either an integer $k$ or the quoted name of a random distribution law (see also 2.24.25, 2.37.36 and **??**).
`randvector` returns a vector of size $n$ containing random integers uniformally distributed between -99 and +99 (default), or between 0 and $k-1$ or containing random integers according to the law put between quotes.
Input :

```
randvector(3)
```

Output :

```
[-54,78,-29]
```

Input :

```
randvector(3,5)
```

Or input :

```
randvector(3,'rand(5)')
```

Output :

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

Input :

```
randvector(3,'randnorm(0,1)')
```

Output :

```
[1.39091705476,-0.136794772167,0.187312440336]
```

Input :

```
randvector(3,2..4)
```

Output :

```
[3.92450003885,3.50059241243,2.7322040787]
```

### 2.37.37 List of differences of consecutive terms : `deltalist`

`deltalist` takes as argument a list.
`deltalist` returns the list of the difference of all pairs of consecutive terms of this list.
Input :

```
deltalist([5,8,1,9])
```

Output :

```
[3,-7,8]
```

### 2.37.38 Make a matrix with a list : `list2mat`

`list2mat` takes as argument a list `l` and an integer `p`.
`list2mat` returns a matrix having `p` column by cutting the list `l` in rows of length `p`. The matrix is filled with `0` if the size of `l` is not a multiple of `p`.
Input :

```
list2mat([5,8,1,9,5,6],2)
```

Output :

```
[[5,8],[1,9],[5,6]]
```

Input :

```
list2mat([5,8,1,9],3)
```

Output :

```
[[5,8,1],[9,0,0]]
```

**Remark**
`Xcas` displays matrix with **[** and **]** and lists with [ and ] as delimiters (the vertical bar of the brackets are thicker for matrices).

### 2.37.39 Make a list with a matrix : `mat2list`

`mat2list` takes as argument a matrix.
`mat2list` returns the list of the coefficients of this matrix.
Input :

```
mat2list([[5,8],[1,9]])
```

Output :

```
[5,8,1,9]
```

## 2.38 Functions for vectors

### 2.38.1 Norms of a vector : `maxnorm l1norm l2norm norm`

The instructions to compute the different norm of a vector are :

- `maxnorm` returns the $l^\infty$ norm of a vector, defined as the maximum of the absolute values of its coordinates.
  Input :

  ```
  maxnorm([3,-4,2])
  ```

  Output :

  ```
  4
  ```

  Indeed : `x=3, y=-4, z=2` and `4=max(|x|,|y|,|z|)`.

- `l1norm` returns the $l^1$ norm of a vector defined as the sum of the absolute values of its coordinates.
  Input :
  ```
  l1norm([3,-4,2])
  ```
  Output :
  ```
  9
  ```
  Indeed : `x=3, y=-4, z=2` and `9=|x|+|y|+|z|`.
- `norm` or `l2norm` returns the $l^2$ norm of a vector defined as the square root of the sum of the squares of its coordinates.
  Input :
  ```
  norm([3,-4,2])
  ```
  Output :
  ```
  sqrt(29)
  ```
  Indeed : `x=3, y=-4, z=2` and $29 = |x|^2 + |y|^2 + |z|^2$.

### 2.38.2 Normalize a vector : `normalize unitV`

`normalize` or `unitV` takes as argument a vector.
`normalize` or `unitV` normalizes this vector for the $l^2$ norm (the square root of the sum of the squares of its coordinates).
Input :

```
normalize([3,4,5])
```

Output :

```
[3/(5*sqrt(2)),4/(5*sqrt(2)),5/(5*sqrt(2))]
```

Indeed : `x=3, y=4, z=5` and $50 = |x|^2 + |y|^2 + |z|^2$.

### 2.38.3 Term by term sum of two lists : `+ .+`

The infixed operator + or `.+` and the prefixed operator `'+'` returns the term by term sum of two lists.
If the two lists do not have the same size, the smallest list is completed with zeros.
Note the difference with sequences : if the infixed operator + or the prefixed operator `'+'` takes as arguments two sequences, it merges the sequences, hence return the sum of all the terms of the two sequences.
Input :

```
[1,2,3]+[4,3,5]
```

Or :

```
[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 (`'+'`).

### 2.38.4 Term by term difference of two lists : `-` `.-`

The infixed operator `-` or `.-` and the prefixed operator `'-'` returns the term by term difference of two lists.
If the two lists do not have the same size, the smallest list is completed with zeros.
Input :

```
[1,2,3]-[4,3,5]
```

Or :

```
[1,2,3] .+ [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 (`'-'`).

### 2.38.5 Term by term product of two lists : `.*`

The infixed operator `.*` returns the term by term product of two lists of the same size.
Input :

```
[1,2,3] .* [4,3,5]
```

Output :

```
[4,6,15]
```

### 2.38.6 Term by term quotient of two lists : `./`

The infixed operator `./` returns the term by term quotient of two lists of the same size.
Input :

```
[1,2,3] ./ [4,3,5]
```

Output :

```
[1/4,2/3,3/5]
```

### 2.38.7 Scalar product : `scalar_product * dotprod dot dotP scalar_Product`

`dot` or `dotP` or `dotprod` or `scalar_product` or `scalarProduct` or the infixed operator `*` takes as argument two vectors.
`dot` or `dotP` or `dotprod` or `scalar_product` or `scalarProduct` or `*` returns the scalar product of these two vectors.
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*4+2*3+3*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[...]`.

### 2.38.8 Cross product : `cross crossP crossproduct`

`cross` or `crossP` or `crossproduct` takes as argument two vectors.
`cross` or `crossP` or `crossproduct` returns the cross product of these two vectors.
Input :

```
cross([1,2,3],[4,3,2])
```

Output :

```
[-5,10,-5]
```

Indeed : $-5 = 2 * 2 - 3 * 3$, $10 = -1 * 2 + 4 * 3$, $-5 = 1 * 3 - 2 * 4$.

## 2.39 Statistic functions : `mean,variance,stddev, stddevp,median,qua`

The functions described here may be used if the statistic serie is contained in a list. See also section 2.42.31 for matrices and chapter **??** for weighted lists.

- `mean` computes the arithmetic mean of a list
  Input :

  ```
  mean([3,4,2])
  ```

  Output :

  ```
  3
  ```

  Input :

  ```
  mean([1,0,1])
  ```

  Output

  ```
  2/3
  ```

- `stddev` computes the standard deviation of a population, if the argument is the population
  Input :

  ```
  stddev([3,4,2])
  ```

  Output :

  ```
  sqrt(2/3)
  ```

- `stddevp` computes an unbiaised estimate of the standard deviation of the population, if the argument is a sample. The following relation holds:

  ```
  stddevp(l)^2=size(l)*stddev(l)^2/(size(l)-1).
  ```

Input :

```
stddevp([3,4,2])
```

Output :

```
1
```

- `variance` computes the variance of a list, that is the square of `stddevp`
  Input :

  ```
  variance([3,4,2])
  ```

  Output :

  ```
  2/3
  ```

- `median` computes the median of a list.
  Input :

  ```
  median([0,1,3,4,2,5,6])
  ```

  Output :

  ```
  3.0
  ```

- `quantile` computes the deciles of a list given as first argument, where the decile is the second argument.
  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 :

```
[4.0]
```

- `quartiles` computes the minimum, the first quartile, the median, the third quartile and the maximum of a list.
  Input :

  ```
  quartiles([0,1,3,4,2,5,6])
  ```

  Output :

  ```
  [[0.0],[1.0],[3.0],[4.0],[6.0]]
  ```

- `boxwhisker` draws the whisker box of a statistic serie stored in a list.
  Input :

  ```
  boxwhisker([0,1,3,4,2,5,6])
  ```

  Output

  ```
  the graph of the whisker box of this statistic
                        list
  ```

**Example**
Define the list `A` by:

```
A:=[0,1,2,3,4,5,6,7,8,9,10,11]
```

Outputs :

1. `11/2` for `mean(A)`
2. `sqrt(143/12)` for `stddev(A)`
3. `0` for `min(A)`
4. `[1.0]` for `quantile(A,0.1)`
5. `[2.0]` for `quantile(A,0.25)`
6. `[5.0]` for `median(A)` or for `quantile(A,0.5)`
7. `[8.0]` for `quantile(A,0.75)`
8. `[9.0]` for `quantile(A,0.9)`
9. `11` for `max(A)`
10. `[[0.0],[2.0],[5.0],[8.0],[11.0]]` for `quartiles(A)`

## 2.40 Table with string as index : `table`

A table is an associative container (or map), it is used to store informations associated to indexes which are much more general than integers, like strings or sequences. It may be used for example to store a table of phone numbers indexed by names.
In `Xcas`, the indexes 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.)
`table` takes as argument a list or a sequence of equalities `index_name=element_value`.
`table` returns this table.
Input :

```
T:=table(3=-10,"a"=10,"b"=20,"c"=30,"d"=40)
```

Input :

```
T["b"]
```

Output :

```
20
```

Input :

```
T[3]
```

Output :

```
-10
```

**Remark**
If you assign `T[n]:= ...` where `T` is a variable name and `n` an integer

- if the variable name was assigned to a list or a sequence, then the $n$-th element of `T` is modified,
- if the variable name was not assigned, a table `T` is created with one entry (corresponding to the index $n$). Note that after the assignation `T` is not a list, despite the fact that $n$ was an integer.

## 2.41 Usual matrix

A matrix is represented by a list of lists, all having the same size. In the `Xcas` answers, the matrix delimiters are **[]** (bold brackets). For example, **[**1,2,3**]** is the matrix [[1,2,3]] with only one row, unlike [1,2,3] (normal brackets) which is the list [1,2,3].
In this document, the input notation ([[1,2,3]]) will be used for input and output.

### 2.41.1 Identity matrix : `idn identity`

`idn` takes as argument an integer $n$ or a square matrix.
`idn` returns the identity matrice of size $n$ or of the same size as the matrix argument.
Input :

```
idn(2)
```

Output :

```
[[1,0],[0,1]]
```

Input :

```
idn(3)
```

Output :

```
[[1,0,0],[0,1,0],[0,0,1]]
```

### 2.41.2 Zero matrix : `newMat matrix`

`newMat(n,p)` or `matrix(n,p)` takes as argument two integers.
`newMat(n,p)` returns the zero matrix with `n` rows and `p` columns.
Input :

```
newMat(4,3)
```

Output :

```
[[0,0,0],[0,0,0],[0,0,0],[0,0,0]]
```

### 2.41.3 Random matrix : `ranm randMat randmatrix`

`ranm` or `randMat` or `randmatrix` takes as argument an integer $n$ or two integers $n, m$ and optionnally a third argument, either an integer $k$ or the quoted name of a random distribution law (see also 2.24.25, 2.37.36 and **??**).
`ranm` returns a vector of size $n$ or a matrix of size $n \times m$ containing random integers uniformally distributed between -99 and +99 (default), or between 0 and $k-1$ or a matrix of size $n \times m$ containing random integers according to the law put between quotes.
Input :

```
ranm(3)
```

Output :

```
[-54,78,-29]
```

Input :

```
ranm(2,4)
```

Output :

```
[[27,-29,37,-66],[-11,76,65,-33]]
```

Input :

```
ranm(2,4,3)
```

Or input :

```
ranm(2,4,'rand(3)')
```

Output :

```
[[0,1,1,0],[0,1,2,0]]
```

Input :

```
ranm(2,4,'randnorm(0,1)')
```

Output :

```
[[1.83785427742,0.793007112053,-0.978388964902,-1.88602023857],
[-1.50900874199,-0.241173369698,0.311373795585,-0.532752431454]]
```

Input :

```
ranm(2,4,2..4)
```

Output :

```
[[2.00549363438,3.03381264955,2.06539073586,2.04844321217],
[3.88383254968,3.28664474655,3.76909781061,2.39113253355]]
```

### 2.41.4 Diagonal of a matrix or matrix of a diagonal : `BlockDiagonal` `diag`

`diag` or `BlockDiagonal` takes as argument a matrix $A$ or a list $l$.
`diag` returns the diagonal of $A$ or the diagonal matrix with the list $l$ on the diagonal (and 0 elsewhere).
Input :

```
diag([[1,2],[3,4]])
```

Output :

```
[1,4]
```

Input :

```
diag([1,4])
```

Output :

```
[[1,0],[0,4]]
```

### 2.41.5 Jordan bloc : `JordanBlock`

`JordanBlock` takes as argument an expression $a$ and an integer $n$.
`JordanBlock` returns a square matrix of size $n$ with $a$ on the principal diagonal, 1 above this diagonal and 0 elsewhere.
Input :

```
JordanBlock(7,3)
```

Output :

```
[[7,1,0],[0,7,1],[0,0,7]]
```

### 2.41.6 Hilbert matrix : `hilbert`

`hilbert` takes as argument an integer $n$.
`hilbert` returns the Hilbert matrix.
A Hilbert matrix is a square matrix of size $n$ whose elements $a_{j,k}$ are :

$$a_{j,k} = \frac{1}{j+k+1}, \quad 0 \le j, 0 \le k$$

Input :

```
hilbert(4)
```

Output :

```
[[1,1/2,1/3,1/4],[1/2,1/3,1/4,1/5],[1/3,1/4,1/5,1/6],
                  [1/4,1/5,1/6,1/7]]
```

### 2.41.7 Vandermonde matrix : `vandermonde`

`vandermonde` takes as argument a vector whose components are denoted by $x_j$ for $j = 0..n-1$.
`vandermonde` returns the corresponding Vandermonde matrix (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`.
Input :

```
vandermonde([a,2,3])
```

Output (if `a` is symbolic else purge(a)) :

```
[[1,1,1],[a,2,3],[a*a,4,9]]
```

## 2.42 Arithmetic and matrix

### 2.42.1 Evaluate a matrix : `evalm`

`evalm` 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`.

### 2.42.2 Addition and substraction of two matrices : `+ - .+ .-`

The infixed operator + or `.+` (resp − or `.-`) are used for the addition (resp substraction) of two matrices.
Input :

```
[[1,2],[3,4]] + [[5,6],[7,8]]
```

Output :

```
[[6,8],[10,12]]
```

Input :

```
[[1,2],[3,4]] - [[5,6],[7,8]]
```

Output :

```
[[-4,-4],[-4,-4]]
```

**Remark**
+ can be used as a prefixed operator, in that case + must be quoted (`'+'`).
Input :

```
'+'([[1,2],[3,4]],[[5,6],[7,8]],[[2,2],[3,3]])
```

Output :

```
[[8,10],[13,15]]
```

### 2.42.3 Multiplication of two matrices : `* &*`

The infixed operator `*` (or `&*`) is used for the multiplication of two matrices.
Input :

```
[[1,2],[3,4]] * [[5,6],[7,8]]
```

Or :

```
[[1,2],[3,4]] &* [[5,6],[7,8]]
```

Output :

```
[[19,22],[43,50]]
```

### 2.42.4 Addition of elements of a column of a matrix : `sum`

`sum` takes as argument a matrix $A$.
`sum` returns the list whose elements are the sum of the elements of each column of the matrix $A$.
Input :

```
sum([[1,2],[3,4]])
```

Output :

```
[4,6]
```

### 2.42.5 Cumulated sum of elements of each column of a matrix : `cumSum`

`cumSum` takes as argument a matrix $A$.
`cumSum` returns the matrix whose columns are the cumulated sum of the elements of the corresponding column of the matrix $A$.
Input :

```
cumSum([[1,2],[3,4],[5,6]])
```

Output :

```
[[1,2],[4,6],[9,12]]
```

since the cumulated sum are : 1, 1+3=4, 1+3+5=9 and 2, 2+4=6, 2+4+6=12.

### 2.42.6 Multiplication of elements of each column of a matrix : `product`

`product` takes as argument a matrix $A$.
`product` returns the list whose elements are the product of the elements of each column of the matrix $A$ (see also 2.37.31 and 2.42.8).
Input :

```
product([[1,2],[3,4]])
```

Output :

```
[3,8]
```

### 2.42.7 Power of a matrix : ^ &^

The infixed operator `^` (or `&^`) is used to raise a matrix to an integral power.
Input :

```
[[1,2],[3,4]] ^ 5
```

Or :

```
[[1,2],[3,4]] &^ 5
```

Output :

```
[[1069,1558],[2337,3406]]
```

### 2.42.8 Hadamard product : `hadamard product`

`hadamard` (or `product`) takes as arguments two matrices $A$ and $B$ of the same size.
`hadamard` (or `product`) returns the matrix where each term is the term by term product of $A$ and $B$.
Input :

```
hadamard([[1, 2],[3,4]],[[5, 6],[7, 8]])
```

Output :

```
[[5,12],[21,32]]
```

See also 2.37.31 and 2.42.6 for `product`.

### 2.42.9 Hadamard product (infixed version): `.*`

`.*` takes as arguments two matrices or two lists $A$ and $B$ of the same size.
`.*` is an infixed operator that returns the matrix or the list where each term is the term by term product of the corresponding terms of $A$ and $B$.
Input :

```
[[1, 2],[3,4]] .* [[5, 6],[7, 8]]
```

Output :

```
[[5,12],[21,32]]
```

Input :

```
[1,2,3,4] .* [5,6,7,8]
```

Output :

```
[5,12,21,32]
```

### 2.42.10 Hadamard division (infixed version): `./`

`./` takes as arguments two matrices or two lists $A$ and $B$ of the same size.
`./` is an infixed operator that returns the matrix or the list where each term is the term by term division of the corresponding terms of $A$ and $B$.
Input :

```
[[1, 2],[3,4]] ./ [[5, 6],[7, 8]]
```

Output :

```
[[1/5,1/3],[3/7,1/2]]
```

### 2.42.11 Hadamard power (infixed version): `.^`

`.^` takes as arguments a matrix or a list $A$ and a real $b$.
`.^` is an infixed operator that returns the matrix or the list where each term is the corresponding term of $A$ raised to the power $b$.
Input :

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

Output :

```
[[1,4],[9,16]]
```

### 2.42.12 Extracting element(s) of a matrix : `[]` `at`

Recall that a matrix is a list of lists with same size.
Input :

```
A:=[[3,4,5],[1,2,6]]
```

Output :

```
[[3,4,5],[1,2,6]]
```

The prefixed function `at` or the index notation `[...]` is used to access to an element or a row or a column of a matrix:

- To extract an element, put the matrix and then, beetween square brackets put its row index, a comma, and its column index. In `Xcas` mode the first index is 0, in other modes the first index is 1.
  Input :

  ```
  [[3,4,5],[1,2,6]][0,1]
  ```

  or

  ```
  A[0,1]
  ```

  or

  ```
  A[0][1]
  ```

  or

  ```
  at(A,[0,1])
  ```

  Output :

  ```
  4
  ```

- To extract a row of the matrix `A`, put the matrix and then, beetween square brackets put the row index, input :

  ```
  [[3,4,5],[1,2,6]][0]
  ```

  or

  ```
  A[0]
  ```

  or

  ```
  at(A,0)
  ```

  Output :

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

- To extract a part of a row, put two arguments beetween the square brackets : the row index and an interval to design the selected columns.
  Input :

  ```
  A[1,0..2]
  ```

  Output :

  ```
  [1,2,6]
  ```

  Input :

  ```
  A[1,1..2]
  ```

  Output :

  ```
  [2,6]
  ```

- To extract a column of the matrix `A`, first tranpose `A` (`transpose(A)`) then extract the row like above.
  Input :

  ```
  tran(A)[1]
  ```

  or

  ```
  at(tran(A),1)
  ```

  Output :

  ```
  [4,2]
  ```

- To extract a part of a column of the matrix `A` as a list, put two arguments beetween the square brackets : an index interval to design the selected rows and the column index.
  Input :

  ```
  A[0..0,1]
  ```

  Output :

  ```
  [4]
  ```

  This may be used to extract a full column, by specifying all the rows as index interval.
  Input :

```
A[0..1,1]
```

Output :

```
[4,2]
```

- To extract a sub-matrix of a matrix, put between the square brackets two intervals : one interval for the selected rows and one interval for the selected columns.
  To definie the matrix `A`, input :

  ```
  A:=[[3,4,5],[1,2,6]]
  ```

  Input :

  ```
  A[0..1,1..2]
  ```

  Output :

  ```
  [[4,5],[2,6]]
  ```

  Input :

  ```
  A[0..1,1..1]
  ```

  Output :

  ```
  [[4],[2]]
  ```

  **Remark** If the second interval is omitted, the sub-matrix is made with the consecutive rows given by the first interval.
  Input :

  ```
  A[1..1]
  ```

  Output :

  ```
  [[1,2,6]]
  ```

You may also assign an element of a matrix using the index notation, if you assign with := a new copy of the matrix is created and the element is modified, if you assign with =<, the matrix is modified in place.

### 2.42.13 Modify an element or a row of a matrix : `subsop`

`subsop` modifies an element or a row of a matrix. It is used mainly for `Maple` and `MuPAD` compatibility. Unlike := or =<, it does not require to store the matrix in a variable.
`subsop` takes two or three arguments, **these arguments are permuted** in `Maple` mode.

1. Modify an element

   - In `Xcas` mode, the first index is 0
     `subsop` has two (resp three) arguments: a matrix `A` and an equality `[r,c]=v` (resp a matrix `A`, a list of index `[r,c]`, a value `v`).
     `subsop` replaces the element `A[r,c]` by `v`.
     Input in `Xcas` mode :
     ```
     subsop([[4,5],[2,6]],[1,0]=3)
     ```
     Or :
     ```
     subsop([[4,5],[2,6]],[1,0],3)
     ```
     Output :
     ```
     [[4,5],[3,6]]
     ```
     **Remark**
     If the matrix is stored in a variable, for example `A:=[[4,5],[2,6]]`, it is easier to input `A[1,0]:=3` which modifies `A` into the matrix `[[4,5],[3,6]]`.
   - In `Mupad, TI` mode, the first index is 1
     `subsop` has two (resp three) arguments: a matrix `A` and an equality `[r,c]=v` (resp a matrix `A`, a list of index `[r,c]`, a value `v`).
     `subsop` replaces the element `A[r,c]` by `v`.
     Input in `Mupad, TI` mode :
     ```
     subsop([[4,5],[2,6]],[2,1]=3)
     ```
     Or :
     ```
     subsop([[4,5],[2,6]],[2,1],3)
     ```
     Output :
     ```
     [[4,5],[3,6]]
     ```
     **Remark**
     If the matrix is stored in a variable, for example `A:=[[4,5],[2,6]]`, it is easier to input `A[2,1]:=3` which modifies `A` into the matrix `[[4,5],[3,6]]`.
   - In `Maple` mode, the arguments are permuted and the first index is 1
     `subsop` has two arguments: an equality `[r,c]=v` and a matrix `A`.
     `subsop` replaces the element `A[r,c]` by `v`.
     Input in `Maple` mode
     ```
     subsop([2,1]=3,[[4,5],[2,6]])
     ```

Output :

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

**Remark**
If the matrix is stored in a variable, for example `A:=[[4,5],[2,6]]`, it is easier to input `A[2,1]:=3` which modifies `A` into the matrix `[[4,5],[3,6]]`.

2. Modify a row

   - in `Xcas` mode, the first index is 0
     `subsop` takes two arguments : a matrix and an equality (the index of the row to be modified, the = sign and the new row value).
     Input in `Xcas` mode :

     ```
     subsop([[4,5],[2,6]],1=[3,3])
     ```

     Output :

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

     **Remark**
     If the matrix is stored in a variable, for example `A:=[[4,5],[2,6]]`, is is easier to input `A[1]:=[3,3]` which modifies `A` into the matrix `[[4,5],[3,3]]`.
   - In `Mupad, TI` mode, the first index is 1
     `subsop` takes two arguments : a matrix and an equality (the index of the row to be modified, the = sign and the new row value).
     Input in `Mupad, TI` mode :

     ```
     subsop([[4,5],[2,6]],2=[3,3])
     ```

     Output :

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

     **Remark**
     If the matrix is stored in a variable, for example `A:=[[4,5],[2,6]]`, it is easier to input `A[2]:=[3,3]` which modifies `A` into the matrix `[[4,5],[3,3]]`.
   - in `Maple` mode, the arguments are permuted and the first index is 1 :
     `subsop` takes two arguments : an equality (the index of the row to be modified, the = sign and the new row value) and a matrix.
     Input in `Maple` mode :

     ```
     subsop(2=[3,3],[[4,5],[2,6]])
     ```

     Output :

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

     **Remark**
     If the matrix is stored in a variable, for example `A:=[[4,5],[2,6]]`, it is easier to input `A[2]:=[3,3]` which modifies `A` into the matrix `[[4,5],[3,3]]`.

**Remark**
Note also that `subsop` with a `'n=NULL'` argument deletes row number `n`. In `Xcas` mode input :

```
subsop([[4,5],[2,6]],'1=NULL')
```

Output :

```
[[4,5]]
```

### 2.42.14 Extract rows or columns of a matrix (Maple compatibility) : `row col`

`row` (resp `col`) extract one or several rows (resp columns) of a matrix.
`row` (resp `col`) takes 2 arguments : a matrix $A$, and an integer $n$ or an interval $n_1..n_2$.
`row` (resp `col`) returns the row (resp column) of index $n$ of $A$, or the sequence of rows (resp columns) of index from $n_1$ to $n_2$ of $A$.
Input :

```
row([[1,2,3],[4,5,6],[7,8,9]],1)
```

Output :

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

Input :

```
row([[1,2,3],[4,5,6],[7,8,9]],0..1)
```

Output :

```
([1,2,3],[4,5,6])
```

Input :

```
col([[1,2,3],[4,5,6],[7,8,9]],1)
```

Output :

```
[2,5,8]
```

Input :

```
col([[1,2,3],[4,5,6],[7,8,9]],0..1)
```

Output :

```
([1,4,7,[2,5,8])
```

### 2.42.15 Remove rows or columns of a matrix : `delrows delcols`

`delrows` (resp `delcols`) removes one or several rows (resp columns) of a matrix.
`delrows` (resp `delcols`) takes 2 arguments : a matrix $A$, and an interval $n_1..n_2$.
`delrows` (resp `delcols`) returns the matrix where the rows (resp columns) of index from $n_1$ to $n_2$ of $A$ are removed.
Input :

```
delrows([[1,2,3],[4,5,6],[7,8,9]],1..1)
```

Output :

```
[[1,2,3],[7,8,9]]
```

Input :

```
delrows([[1,2,3],[4,5,6],[7,8,9]],0..1)
```

Output :

```
[[7,8,9]]
```

Input :

```
delcols([[1,2,3],[4,5,6],[7,8,9]],1..1)
```

Output :

```
[[1,3],[4,6],[7,9]]
```

Input :

```
delcols([[1,2,3],[4,5,6],[7,8,9]],0..1)
```

Output :

```
[[3],[6],[9]]
```

### 2.42.16 Extract a sub-matrix of a matrix (TI compatibility) : `subMat`

`subMat` takes 5 arguments : a matrix $A$, and 4 integers $nl1, nc1, nl2, nc2$, where $nl1$ is the index of the first row, $nc1$ is the index of the first column, $nl2$ is the index of the last row and $nc2$ is the index of the last column.
`subMat(A,nl1,nc1,nl2,nc2)` extract the sub-matrix of the matrix `A` with first element `A[nl1,nc1]` and last element `A[nl2,nc2]`.
Define the matrix `A` :

```
A:=[[3,4,5],[1,2,6]]
```

Input :

```
subMat(A,0,1,1,2)
```

Output :

```
[[4,5],[2,6]]
```

Input :

```
subMat(A,0,1,1,1]
```

Output :

```
[[4],[2]]
```

By default $nl1 = 0$, $nc1 = 0$, $nl2$=`nrows(A)`-1 and $nc2$=`ncols(A)`-1
Input :

```
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]]
```

### 2.42.17 Add a row to another row : `rowAdd`

`rowAdd` takes three arguments : a matrix $A$ and two integers $n1$ and $n2$.
`rowAdd` returns the matrix obtained by replacing in $A$, the row of index $n2$ by the sum of the rows of index $n1$ and $n2$.
Input :

```
rowAdd([[1,2],[3,4]],0,1)
```

Output :

```
[[1,2],[4,6]]
```

### 2.42.18 Multiply a row by an expression : `mRow`

`mRow` takes three arguments : an expression, a matrix $A$ and an integer $n$.
`mRow` returns the matrix obtained by replacing in $A$, the row of index $n$ by the product of the row of index $n$ by the expression.
Input :

```
mRow(12,[[1,2],[3,4]],1)
```

Output :

```
[[1,2],[36,48]]
```

### 2.42.19 Add $k$ times a row to an another row : `mRowAdd`

`mRowAdd` takes four arguments : a real $k$, a matrix $A$ and two integers $n1$ and $n2$.
`mRowAdd` returns the matrix obtained by replacing in $A$, the row of index $n2$ by the sum of the row of index $n2$ and $k$ times the row of index $n1$.
Input :

```
mRowAdd(1.1,[[5,7],[3,4],[1,2]],1,2)
```

Output :

```
[[5,7],[3,4],[4.3,6.4]]
```

### 2.42.20 Exchange two rows : `rowSwap`

`rowSwap` takes three arguments : a matrix $A$ and two integers $n1$ and $n2$.
`rowSwap` returns the matrix obtained by exchanging in $A$, the row of index $n1$ with the row of index $n2$.
Input :

```
rowSwap([[1,2],[3,4]],0,1)
```

Output :

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

### 2.42.21 Make a matrix with a list of matrix : `blockmatrix`

`blockmatrix` takes as arguments two integers $n, m$ and a list of size $n * m$ of matrices of the same dimension $p \times q$ (or more generally such that the $m$ first matrices have the same number of rows and $c$ columns, the $m$ next rows have the same number of rows and $c$ columns, and so on ...). In both cases, we have $n$ blocks of $c$ columns.
`blockmatrix` returns a matrix having $c$ columns by putting these $n$ blocks one under another (vertical gluing). If the matrix arguments have the same dimension $p \times q$, the answer is a matrix of dimension $p * n \times q * m$.
Input :

```
blockmatrix(2,3,[idn(2),idn(2),idn(2),
      idn(2),idn(2),idn(2)])
```

Output :

```
[[1,0,1,0,1,0],[0,1,0,1,0,1],
 [1,0,1,0,1,0],[0,1,0,1,0,1]]
```

Input :

```
blockmatrix(3,2,[idn(2),idn(2),
 idn(2),idn(2),idn(2),idn(2)])
```

Output :

```
           [[1,0,1,0],[0,1,0,1],
[1,0,1,0],[0,1,0,1],[1,0,1,0],[0,1,0,1]]
```

Input :

```
blockmatrix(2,2,[idn(2),newMat(2,3),
        newMat(3,2),idn(3)])
```

Output :

```
[[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]]
```

Input :

```
   blockmatrix(3,2,[idn(1),newMat(1,4),
newMat(2,3),idn(2),newMat(1,2),[[1,1,1]]])
```

Output :

```
[[1,0,0,0,0],[0,0,0,1,0],[0,0,0,0,1],[0,0,1,1,1]]
```

Input :

```
A:=[[1,1],[1,1]];B:=[[1],[1]]
```

then :

```
blockmatrix(2,3,[2*A,3*A,4*A,5*B,newMat(2,4),6*B])
```

Output :

```
[[2,2,3,3,4,4],[2,2,3,3,4,4],
 [5,0,0,0,0,6],[5,0,0,0,0,6]]
```

### 2.42.22 Make a matrix from two matrices : `semi_augment`

`semi_augment` concat two matrices with the same number of columns.
Input :

```
semi_augment([[3,4],[2,1],[0,1]],[[1,2],[4,5]])
```

Output :

```
[[3,4],[2,1],[0,1],[1,2],[4,5]]
```

Input :

```
semi_augment([[3,4,2]],[[1,2,4]])
```

Output :

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

Note the difference with `concat`.
Input :

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

Output :

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

Indeed, when the two matrix $A$ and $B$ have the same dimension, `concat` makes a matrix with the same number of rows than $A$ and $B$ by gluing them side by side.
Input :

```
concat([[3,4],[2,1],[0,1]],[[1,2],[4,5]]
```

Output :

```
[[3,4],[2,1],[0,1],[1,2],[4,5]]
```

but :

```
concat([[3,4],[2,1]],[[1,2],[4,5]]
```

Output :

```
[[3,4,1,2],[2,1,4,5]]
```

### 2.42.23 Make a matrix from two matrices : `augment concat`

`augment` or `concat` concats two matrices $A$ and $B$ having the same number of rows, or having the same number of columns. In the first case, it returns a matrix having the same number of rows than $A$ and $B$ by horizontal gluing, in the second case it returns a matrix having the same number of columns by vertical gluing.
Input :

```
augment([[3,4,5],[2,1,0]],[[1,2],[4,5]])
```

Output :

```
[[3,4,5,1,2],[2,1,0,4,5]]
```

Input :

```
augment([[3,4],[2,1],[0,1]],[[1,2],[4,5]])
```

Output :

```
[[3,4],[2,1],[0,1],[1,2],[4,5]]
```

Input :

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

Output :

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

Note that if $A$ and $B$ have the same dimension, `augment` makes a matrix with the same number of rows than $A$ and $B$ by horizontal gluing, in that case you must use `semi_augment` for vertical gluing.
Input :

```
augment([[3,4],[2,1]],[[1,2],[4,5]])
```

Output :

```
[[3,4,1,2],[2,1,4,5]]]
```

### 2.42.24 Build a matrix with a function : `makemat`

`makemat` takes three arguments :

- a function of two variables `j` and `k` which should return the value of $a_{j,k}$, the element of row index `j` and column index `k` of the matrix to be built.
- two integers $n$ and $p$.

`makemat` returns the matrix $A = (a_{j,k})$ ($j = 0..n-1$ and $k = 0..p-1$) of dimension $n \times p$.
Input :

```
makemat((j,k)->j+k,4,3)
```

or first define the $h$ function:

```
h(j,k):=j+k
```

then, input:

```
makemat(h,4,3)
```

Output :

```
[[0,1,2],[1,2,3],[2,3,4],[3,4,5]]
```

Note that the indices are counted starting from 0.

### 2.42.25 Define a matrix : `matrix`

`matrix` takes three arguments :

- two integers $n$ and $p$.
- a function of two variables `j` and `k` which should return the value of $a_{j,k}$, the element of row index `j` and column index `k` of the matrix to be build.

`matrix` returns the matrix $A = (a_{j,k})$ ($j = 1..n$ and $k = 1..p$) of dimension $n \times p$.
Input :

```
matrix(4,3,(j,k)->j+k)
```

or first define the $h$ function:

```
h(j,k):=j+k
```

then, input:

```
matrix(4,3,h)
```

Output :

```
[[2,3,4],[3,4,5],[4,5,6],[5,6,7]]
```

Note the argument order and the fact that the indices are counted starting from 1. If the last argument is not provided, it defaults to 0.

### 2.42.26 Append a column to a matrix : `border`

`border` takes as argument a matrix `A` of dimension $p*q$ and a list `b` of size $p$ (i.e. `nrows(A)=size(b)`).
`border` returns the matrix obtained by appending `tran(b)` as last column to the matrix `A`, therefore:

```
border(A,b)=tran([op(tran(A)),b])=tran(append(tran(A),b))
```

Input :

```
border([[1,2,4],[3,4,5]],[6,7])
```

Output :

```
[[1,2,4,6],[3,4,5,7]]
```

Input :

```
border([[1,2,3,4],[4,5,6,8],[7,8,9,10]],[1,3,5])
```

Output :

```
[[1,2,3,4,1],[4,5,6,8,3],[7,8,9,10,5]]
```

### 2.42.27 Count the elements of a matrix verifying a property : `count`

`count` takes as arguments : a real function `f` and a real matrix `A` of dimension `p*q` (resp a list `l` of size `n`).
`count` returns `f(A[0,0])+..f(A[p-1,q-1])` (resp `f(l[0])+..f(l[n-1])`)
Hence, if `f` is a boolean function, `count` returns the number of elements of the matrix `A` (resp the list `l`) verifying the property `f`.
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
```

### 2.42.28 Count the elements equal to a given value : `count_eq`

`count_eq` takes as arguments: a real and a real list or a real matrix.
`count_eq` returns the number of elements of the list or matrix equal to the first argument.
Input :

```
count_eq(12,[[2,12,45],[3,7,78]])
```

Output :

```
1
```

### 2.42.29 Count the elements smaller than a given value : `count_inf`

`count_inf` takes as arguments: a real and a real list or a real matrix.
`count_inf` returns the number of elements of the list or matrix which are strictly inferior to the first argument.
Input :

```
count_inf(12,[2,12,45,3,7,78])
```

Output :

```
3
```

### 2.42.30 Count the elements greater than a given value : `count_sup`

`count_sup` takes as arguments: a real and a real list or a real matrix.
`count_sup` returns the number of elements of the list or matrix which are strictly greater to the first argument.
Input :

```
count_sup(12,[[2,12,45],[3,7,78]])
```

Output :

```
2
```

### 2.42.31 Statistics functions acting on column matrices : `mean`, `stddev`, `variance`, `median`, `quantile`, `quartiles`, `boxwhisker`

The following functions work on matrices, acting column by column:

- `mean` computes the arithmetic means of the statistical series stored in the columns of a matrix.
  Input :

  ```
  mean([[3,4,2],[1,2,6]])
  ```

  Output is the vector of the means of each column :

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

  Input :

  ```
  mean([[1,0,0],[0,1,0],[0,0,1]])
  ```

  Output

  ```
  [1/3,1/3,1/3]
  ```

- `stddev` computes the standard deviations of the population statistical series stored in the columns of a matrix.
  Input :

```
stddev([[3,4,2],[1,2,6]])
```

Output is the vector of the standard deviations of each column :

```
[1,1,2]
```

- `variance` computes the variances of the statistical series stored in the columns of a matrix.
  Input :

  ```
  variance([[3,4,2],[1,2,6]])
  ```

  Output is the vector of the variance of each column :

  ```
  [1,1,4]
  ```

- `median` computes the medians of the statistical series stored in the columns of a matrix.
  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 is the vector of the median of each column :

  ```
  [3,3,4,4,4,3,4]
  ```

- `quantile` computes the deciles as specified by the second argument of the statistical series stored in the columns of a matrix.
  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 is the vector of the first quartile of each column :

  ```
  [1,1,2,2,1,1,1]
  ```

  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 is the vector of the third quartile of each column :

  ```
  [3,3,4,4,4,3,4]
  ```

- `quartiles` computes the minima, the first quartiles, the medians, the third quartiles and the maxima of the statistical series stored in the columns of a matrix.
  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 is a matrix, it's first row is the minima of each column, it's second row is the fist quartiles of each column, it's third row the medians of each column, it's fourth row the third quartiles of each column and its last row the maxima of each column:

  ```
  [[0,0,1,0,0,0,0],[1,1,2,2,1,1,1], [2,2,3,3,2,2,3],

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

- `boxwhisker` draws the whisker boxes of the statistical series stored in the columns of a matrix .
  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 :

  ```
     the drawing of the whisker boxes of the
  statistical series of each column of the matrix
                     argument
  ```

### 2.42.32 Dimension of a matrix : `dim`

`dim` takes as argument a matrix $A$.
`dim` returns the list of the number of rows and columns of the matrix $A$.
Input :

```
dim([[1,2,3],[3,4,5]])
```

Output :

```
[2,3]
```

### 2.42.33 Number of rows : `rowdim rowDim nrows`

`rowdim` (or `rowDim` or `nrows`) takes as argument a matrix $A$.
`rowdim` (ou `rowDim` or `nrows`) returns the number of rows of the matrix $A$.
Input :

```
rowdim([[1,2,3],[3,4,5]])
```

or

```
nrows([[1,2,3],[3,4,5]])
```

Output :

```
2
```

### 2.42.34 Number of columns : `coldim colDim ncols`

`coldim` (or `colDim` or `ncols`) takes as argument a matrix $A$.
`coldim` (or `colDim` or `ncols`) returns the number of columns of the matrix $A$.
Input :

```
coldim([[1,2,3],[3,4,5]])
```

or

```
ncols([[1,2,3],[3,4,5]])
```

Output :

```
3
```

## 2.43 Linear algebra

### 2.43.1 Transpose of a matrix : `tran transpose`

`tran` or `transpose` takes as argument a matrix $A$.
`tran` or `transpose` returns the transpose matrix of $A$.
Input :

```
tran([[1,2],[3,4]])
```

Output :

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

### 2.43.2 Inverse of a matrix : `inv /`

`inv` takes as argument a square matrix $A$.
`inv` returns the inverse matrix of $A$.
Input :

```
inv([[1,2],[3,4]])
```

or

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

or

```
A:=[[1,2],[3,4]];1/A
```

Output :

```
[[-2,1],[3/2,1/-2]]
```

### 2.43.3 Trace of a matrix : `trace`

`trace` takes as argument a matrix $A$.
`trace` returns the trace of the matrix $A$, that is the sum of the diagonal elements.
Input :

```
trace([[1,2],[3,4]])
```

Output :

```
5
```

### 2.43.4 Determinant of a matrix : `det`

`det` takes as argument a matrix $A$.
`det` returns the determinant of the matrix $A$.
Input :

```
det([[1,2],[3,4]])
```

Output :

```
-2
```

Input :

```
det(idn(3))
```

Output :

```
1
```

### 2.43.5 Determinant of a sparse matrix : `det_minor`

`det_minor` takes as argument a matrix $A$.
`det_minor` returns the determinant of the matrix $A$ computed by expanding the determinant using Laplace algorithm.
Input :

```
det_minor([[1,2],[3,4]])
```

Output :

```
-2
```

Input :

```
det_minor(idn(3))
```

Output :

```
1
```

### 2.43.6 Rank of a matrix : `rank`

`rank` takes as argument a matrix $A$.
`rank` returns the rank of the matrix $A$.
Input :

```
rank([[1,2],[3,4]])
```

Output :

```
2
```

Input :

```
rank([[1,2],[2,4]])
```

Output :

```
1
```

### 2.43.7 Transconjugate of a matrix : `trn`

`trn` takes as argument a matrix $A$.
`trn` returns the transconjugate of $A$ (i.e. the conjugate of the transpose matrix of $A$).
Input :

```
trn([[i, 1+i],[1, 1-i]])
```

Output after simplification:

```
[[-i,1],[1-i,1+i]]
```

### 2.43.8 Equivalent matrix : `changebase`

`changebase` takes as argument a matrix $A$ and a change-of-basis matrix $P$.
`changebase` returns the matrix $B$ such that $B = P^{-1}AP$.
Input :

```
changebase([[1,2],[3,4]],[[1,0],[0,1]])
```

Output :

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

Input :

```
changebase([[1,1],[0,1]],[[1,2],[3,4]])
```

Output :

```
[[-5,-8],[9/2,7]]
```

Indeed :

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}^{-1} * \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} -5 & -8 \\ \frac{9}{2} & 7 \end{bmatrix}$$

.

### 2.43.9 Basis of a linear subspace : `basis`

`basis` takes as argument a list of vectors generating a linear subspace of $\mathbb{R}^n$.
`basis` returns a list of vectors, that is a basis of this linear subspace.
Input :

```
basis([[1,2,3],[1,1,1],[2,3,4]])
```

Output :

```
[[1,0,-1], [0,1,2]]
```

### 2.43.10 Basis of the intersection of two subspaces : `ibasis`

`ibasis` takes as argument two lists of vectors generating two subspaces of $\mathbb{R}^n$.
`ibasis` returns a list of vectors that is a basis of the intersection of these two subspaces.
Input :

```
ibasis([[1,2]],[[2,4]])
```

Output :

```
[[1,2]]
```

### 2.43.11 Image of a linear application : `image`

`image` takes as argument the matrix of an linear application $f$ with respect to the canonical basis.
`image` returns a list of vectors that is a basis of the image of $f$.
Input :

```
image([[1,1,2],[2,1,3],[3,1,4]])
```

Output :

```
[[-1,0,1],[0,-1,-2]]
```

### 2.43.12 Kernel of a linear application : `kernel nullspace ker`

`ker` (or `kernel` or `nullspace`) takes as argument the matrix of an linear application $f$ with respect to the canonical basis.
`ker` (or `kernel` or `nullspace`) returns a list of vectors that is a basis of the kernel of $f$.
Input :

```
ker([[1,1,2],[2,1,3],[3,1,4]])
```

Output :

```
[[1,1,-1]]
```

The kernel is generated by the vector `[1,1,-1]`.

### 2.43.13 Kernel of a linear application : `Nullspace`

**Warning** This function is useful in Maple mode only (hit the state line red button then `Prog style`, then choose Maple and Apply).
`Nullspace` is the inert form of `nullspace`.
`Nullspace` takes as argument an integer matrix of an linear application $f$ with respect to the canonical basis.
`Nullspace`) followed by `mod p` returns a list of vectors that is a basis of the kernel of $f$ computed in $\mathbb{Z}/p\mathbb{Z}[X]$.
Input :

```
Nullspace([[1,1,2],[2,1,3],[3,1,4]])
```

Output :

```
nullspace([[1,1,2],[2,1,3],[3,1,4]])
```

Input (in Maple mode):

```
Nullspace([[1,2],[3,1]]) mod 5
```

Output :

```
[2,-1]
```

In Xcas mode, the equivalent input is :

```
nullspace([[1,2],[3,1]] % 5)
```

Output :

```
[2% 5,-1]
```

### 2.43.14 Subspace generated by the columns of a matrix : `colspace`

`colspace` takes as argument the matrix $A$ of an linear application $f$ with respect to the canonical basis.
`colspace` returns a matrix. The columns of this matrix are a basis of the subspace generated by the columns of $A$.
`colspace` may have a variable name as second argument, where `Xcas` will store the dimension of the subspace generated by the columns of $A$.
Input :

```
colspace([[1,1,2],[2,1,3],[3,1,4]])
```

Output :

```
[[-1,0],[0,-1],[1,-2]]
```

Input :

```
colspace([[1,1,2],[2,1,3],[3,1,4]],dimension)
```

Output :

```
[[-1,0],[0,-1],[1,-2]]
```

Then input:

```
dimension
```

Output :

```
2
```

### 2.43.15 Subspace generated by the rows of a matrix : `rowspace`

`rowspace` takes as argument the matrix $A$ of an linear application $f$ with respect to the canonical basis.
`rowspace` returns a list of vectors that is a basis of the subspace generated by the rows of $A$.
`rowspace` may have a variable name as second argument where `Xcas` will store the dimension of the subspace generated by the rows of $A$.
Input :

```
rowspace([[1,1,2],[2,1,3],[3,1,4]])
```

Output :

```
[[-1,0,-1],[0,-1,-1]]
```

Input :

```
rowspace([[1,1,2],[2,1,3],[3,1,4]],dimension)
```

Output :

```
[[-1,0,-1],[0,-1,-1]]
```

Then input:

```
dimension
```

Output :

```
2
```

## 2.44 Linear Programmation

Linear programming problems are maximization problem of a linear functional under linear equality or inequality constraints. The most simple case can be solved directly by the so-called simplex algorithm. Most cases requires to solve an auxiliary linear programming problem to find an initial vertex for the simplex algorithm.

### 2.44.1 Simplex algorithm: `simplex_reduce`

**The simple case**
The function `simplex_reduce` makes the reduction by the simplex algorithm to find :

$$\max(c.x), \quad A.x \leq b,\ x \geq 0,\ b \geq 0$$

where $c, x$ are vectors of $\mathbb{R}^n$, $b \geq 0$ is a vector of $\mathbb{R}^p$ and $A$ is a matrix of $p$ rows and $n$ columns.
`simplex_reduce` takes as argument `A,b,c` et returns `max(c.x)`, the augmented solution of `x` (augmented since the algorithm works by adding rows($A$) auxiliary variables) and the reduced matrix.

**Example**
Find

$$\max(X+2Y) \text{ where } \left\{\begin{array}{rcl} (X,Y) & \geq & 0 \\ -3X+2Y & \leq & 3 \\ X+Y & \leq & 4 \end{array}\right.$$

Input :

```
simplex_reduce([[-3,2],[1,1]],[3,4],[1,2])
```

Output :

```
7,[1,3,0,0],[[0,1,1/5,3/5,3],[1,0,(-1)/5,2/5,1],
                    [0,0,1/5,8/5,7]]
```

Which 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 :
`[[0,1,1/5,3/5,3],[1,0,(-1)/5,2/5,1], [0,0,1/5,8/5,7]]`.

**A more complicate case that reduces to the simple case**
With the former call of `simplex_reduce`, we 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 :

$$\min(2x+y-z+4) \quad \text{where} \left\{\begin{array}{rcl} x & \leq & 1 \\ y & \geq & 2 \\ x+3y-z & = & 2 \\ 2x-y+z & \leq & 8 \\ -x+y & \leq & 5 \end{array}\right.$$

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 :

$$\left\{\begin{array}{rcl} X & \geq & 0 \\ Y & \geq & 0 \\ 2(1-X)-(Y+2)+5-X+3Y & \leq & 8 \\ -(1-X)+(Y+2) & \leq & 5 \end{array}\right.$$

or to find the minimum of :

$$(-X-2Y+3) \quad \text{where} \left\{\begin{array}{rcl} X & \geq & 0 \\ Y & \geq & 0 \\ -3X+2Y & \leq & 3 \\ X+Y & \leq & 4 \end{array}\right.$$

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. We found `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 vertex must be found before applying the simplex algorithm. Therefore, `simplex_reduce` may be called by specifying this starting vertex, in that case, all the arguments including the starting vertex are grouped in a single matrix.

We first illustrate this kind of call in the simple case where the starting point does not require solving an auxiliary problem. If `A` has $p$ rows and $n$ columns and if we define :

```
B:=augment(A,idn(p)); C:=border(B,b);
d:=append(-c,0$(p+1)); D:=augment(C,[d]);
```

`simplex_reduce` may be called with `D` as single argument.
For the previous example, input :

```
A:=[[-3,2],[1,1]];B:=augment(A,idn(2));
C:=border(B,[3,4]); D:=augment(C,[[-1,-2,0,0,0]])
```

Here `C=[[-3,2,1,0,3],[1,1,0,1,4]]`
and `D=[[-3,2,1,0,3],[1,1,0,1,4],[-1,-2,0,0,0]]`
Input :

```
simplex_reduce(D)
```

Output is the same result as before.

**Back to the general case.**

The standard form of a linear programming problem is similar to the simplest case above, but with $Ax = b$ (instead of $Ax \leq b$) under the conditions $x \geq 0$. We may further assume that $b \geq 0$ (if not, one can change the sign of the corresponding line).

- The first problem is to find an $x$ in the $Ax = b, x \geq 0$ domain. Let $m$ be the number of lines of $A$. Add artificial variables $y_1, ..., y_m$ and maximize $-\sum y_i$ under the conditions $Ax = b, x \geq 0, y \geq 0$ starting with initial value 0 for $x$ variables and $y = b$ (to solve this with `Xcas`, call `simplex_reduce` with a single matrix argument obtained by augmenting $A$ by the identity, $b$ unchanged and an artificial $c$ with 0 under $A$ and 1 under the identity). If the maximum exists and is 0, the identity submatrix above the last column corresponds to an $x$ solution, we may forget the artificial variables (they are 0 if the maximum is 0).

- Now we make a second call to `simplex_reduce` with the original $c$ and the value of $x$ we found in the domain.

- Example : find the minimum of $2x + 3y - z + t$ with $x, y, z, t \geq 0$ and :

  $$\left\{ \begin{array}{rcl} -x - y + t &=& 1 \\ y - z + t &=& 3 \end{array} \right.$$

  This is equivalent to find the opposite of the maximum of $-(2x+3y-z+t)$. Let us add two artificial variables $y_1$ and $y_2$,

```
simplex_reduce([[-1,-1,0,1,1,0,1],
[0,1,-1,1,0,1,3],
[0,0,0,0,1,1,0]])
```

Output: optimum=0, artificial variables=0, and the matrix

$$\begin{pmatrix} -1/2 & 0 & -1/2 & 1 & 1/2 & 1/2 & 2 \\ 1/2 & 1 & -1/2 & 0 & -1/2 & 1/2 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

Columns 2 and 4 are the columns of the identity (in lines 1 and 2). Hence $x = (0, 1, 0, 2)$ is an initial point in the domain. We are reduced to solve the initial problem, after replacing the lines of $Ax = b$ by the two first lines of the answer above, removing the last columns corresponding to the artificial variables. We add $c.x$ as last line

```
simplex_reduce([[-1/2,0,-1/2,1,2],
[1/2,1,-1/2,0,1],[2,3,-1,1,0]])
```

Output: maximum=-5, hence the minimum of the opposite is 5, obtained for $(0, 1, 0, 2)$, after replacement $x = 0$, $y = 1$, $z = 0$ and $t = 2$.

For more details, search google for `simplex algorithm`.

## 2.45 Different matrix norm

### 2.45.1 $l^2$ matrix norm : `nomm l2norm`

`norm` (or `l2norm`) takes as argument a matrix $A = a_{j,k}$ (see also 2.38.1).
`norm` (or `l2norm`) returns $\sqrt{\sum_{j,k} a_{j,k}^2}$.
Input :

```
norm([[1,2],[3,-4]])
```

or

```
l2norm([[1,2],[3,-4]])
```

Output :

```
sqrt(30)
```

### 2.45.2 $l^\infty$ matrix norm : `maxnorm`

`maxnorm` takes as argument a matrix $A = a_{j,k}$ (see also 2.38.1).
`maxnorm` returns $\max(|a_{j,k}|)$.
Input :

```
maxnorm([[1,2],[3,-4]])
```

Output :

```
4
```

### 2.45.3 Matrix row norm : `rownorm rowNorm`

`rownorm` (or `rowNorm`) takes as argument a matrix $A = a_{j,k}$.
`rownorm` (or `rowNorm`) returns $\max_k(\sum_j |a_{j,k}|)$.
Input :

```
rownorm([[1,2],[3,-4]])
```

or

```
rowNorm([[1,2],[3,-4]])
```

Output :

```
7
```

Indeed : $\max(1 + 2, 3 + 4) = 7$

### 2.45.4 Matrix column norm : `colnorm colNorm`

`colnorm` (or `colNorm`) takes as argument a matrix $A = a_{j,k}$.
`colnorm` (or `colNorm`) returns $\max_j(\sum_k(|a_{j,k}|))$.
Input :

```
colnorm([[1,2],[3,-4]])
```

or

```
colNorm([[1,2],[3,-4]])
```

Output :

```
6
```

Indeed : $\max(1 + 3, 2 + 4) = 6$

## 2.46 Matrix reduction

### 2.46.1 Eigenvalues : `eigenvals`

`eigenvals` takes as argument a square matrix $A$ of size $n$.
`eigenvals` returns the sequence of the $n$ eigenvalues of $A$.
**Remark** : If $A$ is exact, `Xcas` may not be able to find the exact roots of the characteristic polynomial, `eigenvals` will return approximate eigenvalues of $A$ if the coefficients are numeric or a subset of the eigenvalues if the coefficients are symbolic.
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 :

```
(0.324869129433,4.21431974338,1.46081112719)
```

### 2.46.2 Eigenvalues : `egvl eigenvalues eigVl`

`egvl` (or `eigenvalues eigVl`) takes as argument a square matrix $A$ of size $n$.
`egvl` (or `eigenvalues eigVl`) returns the Jordan normal form of $A$.
**Remark** : If $A$ is exact, `Xcas` may not be able to find the exact roots of the characteristic polynomial, `eigenvalues` will return an approximate diagonalozation of $A$ if the coefficients are numeric.
Input :

```
egvl([[4,1,-2],[1,2,-1],[2,1,0]])
```

Output :

```
[[2,1,0],[0,2,1],[0,0,2]]
```

Input :

```
egvl([[4,1,0],[1,2,-1],[2,1,0]])
```

Output :

```
[[0.324869129433,0,0],[0,4.21431974338,0],[0,0,1.46081112719]]
```

### 2.46.3 Eigenvectors : `egv eigenvectors eigenvects eigVc`

`egv` (or `eigenvectors eigenvects eigVc`) takes as argument a square matrix $A$ of size $n$.
If $A$ is a diagonalizable matrix, `egv` (or `eigenvectors eigenvects eigVc`) returns a matrix which columns are the eigenvectors of the matrix $A$. Otherwise, it will fail (see also `jordan` for characteristic vectors).
Input :

```
egv([[1,1,3],[1,3,1],[3,1,1]])
```

Output :

```
[[-1,1,1],[2,1,0],[-1,1,-1]]
```

Input :

```
egv([[4,1,-2],[1,2,-1],[2,1,0]])
```

Output :

```
"Not diagonalizable at eigenvalue 2"
```

In complex mode, input :

```
egv([[2,0,0],[0,2,-1],[2,1,2]])
```

Output :

```
[0,1,0],[-1,-2,-1],[i,0,-i]]
```

### 2.46.4 Rational Jordan matrix : `rat_jordan`

`rat_jordan` takes as argument a square matrix $A$ of size $n$ with exact coefficients.
`rat_jordan` returns :

- in `Xcas`, `Mupad` or `TI` mode
  a sequence of two matrix : a matrix $P$ (the columns of $P$ are the eigenvectors if $A$ is diagonalizable in the field of it's coefficients) and the rational Jordan matrix $J$ of $A$, that is the most reduced matrix in the field of the coefficients of $A$ (or the complexified field in complex mode), where

$$J = P^{-1}AP$$

- in `Maple` mode
  the Jordan matrix $J$ of $A$. We can also have the matrix $P$ verifying $J = P^{-1}AP$ in a variable by passing this variable as second argument, for example

```
rat_jordan([[1,0,0],[1,2,-1],[0,0,1]],'P')
```

**Remarks**

- the syntax `Maple` is also valid in the other modes, for example, in `Xcas` mode input

```
rat_jordan([[4,1,1],[1,4,1],[1,1,4]],'P')
```

  Output :

```
[[1,-1,1/2],[1,0,-1],[1,1,1/2]]
```

  then `P` returns

```
[[6,0,0],[0,3,0],[0,0,3]]
```

- the coefficients of $P$ and $J$ belongs to the same field as the coefficients of $A$. For example, in `Xcas` mode, input :

```
rat_jordan([[1,0,1],[0,2,-1],[1,-1,1]])
```

  Output :

```
[[1,1,2],[0,0,-1],[0,1,2]],[[0,0,-1],[1,0,-3],[0,1,4]]
```

  Input (put `-pcar(...)` because the argument of `companion` is an unit polynomial (see 2.46.10)

```
companion(-pcar([[1,0,1],[0,2,-1],[1,-1,1]],x),x)
```

Output :

```
[[0,0,-1],[1,0,-3],[0,1,4]]
```

Input :

```
rat_jordan([[1,0,0],[0,1,1],[1,1,-1]])
```

Output :

```
[[-1,0,0],[1,1,1],[0,0,1]],[[1,0,0],[0,0,2],[0,1,0]]
```

Input :

```
factor(pcar([[1,0,0],[0,1,1],[1,1,-1]],x))
```

Output :

```
-(x-1)*(x^2-2)
```

Input :

```
companion((x^2-2),x)
```

Output :

```
[[0,2],[1,0]]
```

- When $A$ is symetric and has eigenvalues with an multiple order, `Xcas` returns orthogonal eigenvectors (not always of norm equal to 1) i.e. `tran(P)*P` is a diagonal matrix where the diagonal is the square norm of the eigenvectors, for example :

  ```
  rat_jordan([[4,1,1],[1,4,1],[1,1,4]])
  ```

  returns :

  ```
  [[1,-1,1/2],[1,0,-1],[1,1,1/2]],[[6,0,0],[0,3,0],[0,0,3]]
  ```

Input in `Xcas`, `Mupad` or `TI` mode :

```
rat_jordan([[1,0,0],[1,2,-1],[0,0,1]])
```

Output :

```
[[0,1,0],[1,0,1],[0,1,1]],[[2,0,0],[0,1,0],[0,0,1]]
```

Input in `Xcas`, `Mupad` or `TI` mode :

```
rat_jordan([[4,1,-2],[1,2,-1],[2,1,0]])
```

Output :

```
[[[1,2,1],[0,1,0],[1,2,0]],[[2,1,0],[0,2,1],[0,0,2]]]
```

In complex mode and in `Xcas`, `Mupad` or `TI` mode , input :

```
rat_jordan([[2,0,0],[0,2,-1],[2,1,2]])
```

Output :

```
[[1,0,0],[-2,-1,-1],[0,-i,i]],[[2,0,0],[0,2-i,0],[0,0,2+i]]
```

Input in `Maple` mode :

```
rat_jordan([[1,0,0],[1,2,-1],[0,0,1]],'P')
```

Output :

```
[[2,0,0],[0,1,0],[0,0,1]]
```

then input :

```
P)
```

Output :

```
[[0,1,0],[1,0,1],[0,1,1]]]
```

### 2.46.5 Jordan normal form : `jordan`

`jordan` takes as argument a square matrix $A$ of size $n$.
`jordan` returns :

- in `Xcas`, `Mupad` or `TI` mode
  a sequence of two matrix : a matrix $P$ which columns are the eigenvectors and characteristic vectors of the matrix $A$ and the Jordan matrix $J$ of $A$ verifying $J = P^{-1}AP$,
- in `Maple` mode
  the Jordan matrix $J$ of $A$. We can also have the matrix $P$ verifying $J = P^{-1}AP$ in a variable by passing this variable as second argument, for example

  ```
  jordan([[1,0,0],[0,1,1],[1,1,-1]],'P')
  ```

**Remarks**

- the syntax `Maple` is also valid in the other modes, for example, in `Xcas` mode input :

  ```
  jordan([[4,1,1],[1,4,1],[1,1,4]],'P')
  ```

Output :

```
[[1,-1,1/2],[1,0,-1],[1,1,1/2]]
```

then `P` returns

```
[[6,0,0],[0,3,0],[0,0,3]]
```

- When $A$ is symetric and has eigenvalues with an multiple order, `Xcas` returns orthogonal eigenvectors (not always of norm equal to 1) i.e. `tran(P)*P` is a diagonal matrix where the diagonal is the square norm of the eigenvectors, for example :

  ```
  jordan([[4,1,1],[1,4,1],[1,1,4]])
  ```

  returns :

  ```
  [[1,-1,1/2],[1,0,-1],[1,1,1/2]],[[6,0,0],[0,3,0],[0,0,3]]
  ```

Input in `Xcas`, `Mupad` or `TI` mode :

```
jordan([[1,0,0],[0,1,1],[1,1,-1]])
```

Output :

```
[[1,0,0],[0,1,1],[1,1,-1]],[[-1,0,0],[1,1,1],[0,-sqrt(2)-1,sqrt(2)-1]],[[1,0,
```

Input in `Maple` mode :

```
jordan([[1,0,0],[0,1,1],[1,1,-1]])
```

Output :

```
[[1,0,0],[0,-(sqrt(2)),0],[0,0,sqrt(2)]]
```

then input :

```
P)
```

Output :

```
[[-1,0,0],[1,1,1],[0,-sqrt(2)-1,sqrt(2)-1]]
```

Input in `Xcas`, `Mupad` or `TI` mode :

```
jordan([[4,1,-2],[1,2,-1],[2,1,0]])
```

Output :

```
[[[1,2,1],[0,1,0],[1,2,0]],[[2,1,0],[0,2,1],[0,0,2]]]
```

In complex mode and in `Xcas`, `Mupad` or `TI` mode , input :

```
jordan([[2,0,0],[0,2,-1],[2,1,2]])
```

Output :

```
[[1,0,0],[-2,-1,-1],[0,-i,i]],[[2,0,0],[0,2-i,0],[0,0,2+i]]
```

### 2.46.6 Characteristic polynomial : `charpoly`

`charpoly` (or `pcar`) takes one or two argument(s), a square matrix $A$ of size $n$ and optionnally the name of a symbolic variable.
`charpoly` returns the characteristic polynomial $P$ of $A$ written as the list of its coefficients if no variable name was provided or written as an expression with respect to the variable name provided as second argument.
The characteristic polynomial $P$ of $A$ is defined as

$$P(x) = \det(x.I - A)$$

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 `normal(poly2symb([1,-6,12,-8],x))` to get its symbolic representation).
Input :

```
purge(X):; charpoly([[4,1,-2],[1,2,-1],[2,1,0]],X)
```

Output :

```
X^3-6*X^2+12*X-8
```

### 2.46.7 Characteristic polynomial using Hessenberg algorithm : `pcar_hessenberg`

`pcar_hessenberg` takes as argument a square matrix $A$ of size $n$ and optionnaly the name of a symbolic variable.
`pcar_hessenberg` returns the characteristic polynomial $P$ of $A$ written as the list of its coefficients if no variable was provided or written in its symbolic form with respect to the variable name given as second argument, where

$$P(x) = \det(xI - A)$$

The characteristic polynomial is computed using the Hessenberg algorithm (see e.g. Cohen) which is more efficient ($O(n^3)$ deterministic) if the coefficients of $A$ 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 e.g. in $\mathbb{Q}$.
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 :

```
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

$$x^3 - 6x^2 + 12x - 8$$

### 2.46.8 Minimal polynomial : `pmin`

`pmin` takes one (resp two) argument(s): a square matrix $A$ of size $n$ and optionnaly the name of a symbolic variable.
`pmin` returns the minimal polynomial of $A$ written as a list of its coefficients if no variable was provided, or written in symbolic form with respect to the variable name given as second argument. The minimal polynomial of $A$ is the polynomial $P$ having minimal degree such that $P(A) = 0$.
Input :

```
pmin([[1,0],[0,1]])
```

Output :

```
[1,-1]
```

Input :

```
pmin([[1,0],[0,1]],x)
```

Output :

```
x-1
```

Hence the minimal polynomial of [[1,0],[0,1]] 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^2-4*x+4
```

Hence, the minimal polynomial of [[2,1,0],[0,2,0],[0,0,2]] is $x^2 - 4x + 4$.

### 2.46.9 Adjoint matrix : `adjoint_matrix`

`adjoint_matrix` takes as argument a square matrix $A$ of size $n$.
`adjoint_matrix` returns the list of the coefficients of $P$ (the characteristic polynomial of $A$), and the list of the matrix coefficients of $Q$ (the adjoint matrix of $A$).

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 of $A$ is the comatrix of $xI - A$. It is a polynomial of degree $n - 1$ in $x$ having matrix coefficients. The following relation holds:

$$P(x) \times I = \det(xI - A)I = (xI - A)Q(x)$$

Since the polynomial $P(x) \times I - P(A)$ (with matrix coefficients) is also divisible by $x \times I - A$ (by algebraic identities), this proves that $P(A) = 0$. We also have $Q(x) = I \times x^{n-1} + ... + B_0$ where $B_0 =$ is the comatrix of $A$ (up to the sign if $n$ is odd). Input :

```
adjoint_matrix([[4,1,-2],[1,2,-1],[2,1,0]])
```

Output :

```
[ [1,-6,12,-8],
[ [[1,0,0],[0,1,0],[0,0,1]], [[-2,1,-2],
[1,-4,-1],[2,1,-6]], [[1,-2,3],[-2,4,2],[-3,-2,7]] ] ]
```

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 :

$$B = Q(0) = [[1, -2, 3], [-2, 4, 2], [-3, -2, 7]]$$

Hence the inverse of $A$ is equal to :

$$1/8 * [[1, -2, 3], [-2, 4, 2], [-3, -2, 7]]$$

The adjoint matrix of $A$ is :

$$[[x^2-2x+1, x-2, -2x+3], [x-2, x^2-4x+4, -x+2], [2x-3, x-2, x^2-6x+7]]$$

Input :

```
adjoint_matrix([[4,1],[1,2]])
```

Output :

```
[[1,-6,7],[[[1,0],[0,1]],[[-2,1],[1,-4]]]]
```

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

$$Q(0) = -[[-2, 1], [1, -4]]$$

Hence the inverse of $A$ is equal to :

$$-1/7 * [[-2, 1], [1, -4]]$$

The adjoint matrix of $A$ is :

$$-[[x - 2, 1], [1, x - 4]]$$

### 2.46.10 Companion matrix of a polynomial : `companion`

`companion` takes as argument an unitary polynomial $P$ and the name of its variable.
`companion` returns the matrix whose characteristic polynomial is $P$.
If $P(x) = x^n + a_{n-1}x^{n-1} + ... + a_{-1}x + a_0$, this matrix is equal to the unit matrix of size $n-1$ bordered with $[0, 0.., 0, -a_0]$ as first row, and with $[-a_0, -a_1, ...., -a_{n-1}]$ as last column.
Input :

```
companion(x^2+5x-7,x)
```

Output :

```
[[0,7],[1,-5]]
```

Input :

```
companion(x^4+3x^3+2x^2+4x-1,x)
```

Output :

```
[[0,0,0,1],[1,0,0,-4],[0,1,0,-2],[0,0,1,-3]]
```

### 2.46.11 Hessenberg matrix reduction : `hessenberg`

`hessenberg` takes as argument a matrix $A$.
`hessenberg` returns a matrix $B$ equivalent to $A$ where the coefficients below the sub-principal diagonal are zero. $B$ is a Hessenberg matrix.
Input :

```
hessenberg([[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]])
```

Output :

```
[[3,8,5,10,2],[2,1,1/2,-5,-1],[0,2,1,8,2],
      [0,0,1/2,8,1],[0,0,0,-26,-3]]
```

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]] :;
   B:= hessenberg(A):; pcar(A); pcar(B)
```

Output: `[1,-7,-66,-24]`.

### 2.46.12 Hermite normal form : `ihermite`

`ihermite` takes as argument a matrix `A` with coefficients in $\mathbb{Z}$.
`ihermite` returns two matrices `U` and `B` such that `B=U*A`, `U` is invertible in $\mathbb{Z}$ ($\det(U) = \pm 1$) and `B` is upper-triangular. Moreover, the absolute value of the coefficients above the diagonal of `B` are smaller than the pivot of the column divided by 2.

The answer is obtained by a Gauss-like reduction algorithm using only operations of rows with integer coefficients and invertible in $\mathbb{Z}$.
Input :

```
A:=[[9,-36,30],[-36,192,-180],[30,-180,180]];
U,B:=ihermite(A)
```

Output :

```
[[9,-36,30],[-36,192,-180],[30,-180,180]],
[[13,9,7],[6,4,3],[20,15,12]],[[3,0,30],[0,12,0],[0,0,60]]
```

**Application: Compute a $\mathbb{Z}$-basis of the kernel of a matrix having integer coefficients**
Let `M` be a matrix with integer coefficients. Input :

```
(U,A):=ihermite(transpose(M)).
```

This returns $U$ and $A$ such that `A=U*transpose(M)` hence `transpose(A)=M*transpose(U)`.
The columns of `transpose(A)` which are identically 0 (at the right, coming from the rows of $A$ which are identically 0 at the bottom) correspond to columns of `transpose(U)` which form a basis of `Ker(M)`. In other words, the rows of `A` which are identically 0 correspond to rows of `U` which form a basis of `Ker(M)`.
**Example**
Let `M:=[[1,4,7],[2,5,8],[3,6,9]]`. Input

```
U,A:=ihermite(tran(M))
```

Output

```
U:=[[-3,1,0],[4,-1,0],[-1,2,-1]] and
A:=[[1,-1,-3],[0,3,6],[0,0,0]]
```

Since `A[2]=[0,0,0]`, a $\mathbb{Z}$-basis of `Ker(M)` is `U[2]=[-1,2,-1]`.
Verification `M*U[2]=[0,0,0]`.

### 2.46.13 Smith normal form : `ismith`

`ismith` takes as argument a matrix with coefficients in $\mathbb{Z}$.
`ismith` returns three matrices `U,B` and `V` such that `B=U*A*V`, `U` and `V` are invertible in $\mathbb{Z}$, `B` is diagonal, and `B[i,i]` divides `B[i+1,i+1]`. The coefficients `B[i,i]` are called invariant factors, they are used to describe the structure of finite abelian groups.
Input :

```
A:=[[9,-36,30],[-36,192,-180],[30,-180,180]];
U,B,V:=ismith(A)
```

Output :

```
[[-3,0,1],[6,4,3],[20,15,12]],
[[3,0,0],[0,12,0],[0,0,60]],
[[1,24,-30],[0,1,0],[0,0,1]]
```

The invariant factors are 3, 12 and 60.

## 2.47 Isometries

### 2.47.1 Recognize an isometry : `isom`

`isom` takes as argument the matrix of an linear application in dimension 2 or 3.
`isom` returns :

- if the linear application is a direct isometry,
  the list of the characteristic elements of this isometry and `+1`,
- if the linear application is an indirect isometry,
  the list of the characteristic elements of this isometry and `-1`
- if the linear application is not an isometry,
  `[0]`.

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 :

```
isom(sqrt(2)/2*[[1,-1],[1,1]])
```

Output :

```
[pi/4,1]
```

Hence, this isometry is a 2-d rotation of angle $\frac{\pi}{4}$.
Input :

```
isom([[0,0,1],[0,1,0],[0,0,1]])
```

Output :

```
[0]
```

therefore this transformation is not an isometry.

### 2.47.2 Find the matrix of an isometry : `mkisom`

`mkisom` takes as argument :

- In dimension 3, the list of characteristic elements (axis direction, angle for a rotation or normal to the plane for a symmetry) and `+1` for a direct isometry or `-1` an indirect isometry.
- In dimension 2, a characteristic element (an angle or a vector) and `+1` for a direct isometry (rotation) or `-1` for an indirect isometry (symmetry).

`mkisom` returns the matrix of the corresponding isometry.
Input :

```
mkisom([[-1,2,-1],pi],1)
```

Output the matrix of the rotation of axis $[-1, 2, -1]$ and angle $\pi$:

```
[[-2/3,-2/3,1/3],[-2/3,1/3,-2/3],[1/3,-2/3,-2/3]]
```

Input :

```
mkisom([pi],-1)
```

Output the matrix of the symmetry with respect to $O$ :

```
[[-1,0,0],[0,-1,0],[0,0,-1]]
```

Input :

```
mkisom([1,1,1],-1)
```

Output the matrix of the symmetry with respect to the plane $x + y + z = 0$ :

```
[[1/3,-2/3,-2/3],[-2/3,1/3,-2/3],[-2/3,-2/3,1/3]]
```

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 $x + y + z = 0$:

```
[[0,-1,0],[0,0,-1],[-1,0,0]]
```

Input :

```
mkisom(pi/2,1)
```

Output the matrix of the plane rotation of angle $\frac{\pi}{2}$ :

```
[[0,-1],[1,0]]
```

Input :

```
mkisom([1,2],-1)
```

Output matrix of the plane symmetry with respect to the line of equation $x + 2y = 0$:

```
[[3/5,-4/5],[-4/5,-3/5]]
```

## 2.48 Matrix factorizations

Note that most matrix factorization algorithms are implemented numerically, only a few of them will work symbolically.

### 2.48.1 Cholesky decomposition : `cholesky`

`cholesky` takes as argument a square symetric positive definite matrix `M` of size $n$.
`cholesky` returns a symbolic or numeric matrix `P`. `P` is a lower triangular matrix such that :

```
tran(P)*P=M
```

Input :

```
cholesky([[1,1],[1,5]])
```

Output :

```
[[1,0],[1,2]]
```

Input :

```
cholesky([[3,1],[1,4]])
```

Output :

```
[[sqrt(3),0],[(sqrt(3))/3,(sqrt(33))/3]]
```

Input :

```
cholesky([[1,1],[1,4]])
```

Output :

```
[[1,0],[1,sqrt(3)]]
```

**Warning** If the matrix argument $A$ is not a symetric matrix, `cholesky` does not return an error, instead `cholesky` will use the symetric matrix $B$ of the the quadratic form $q$ corresponding to the (non symetric) bilinear form of matrix $A$.
Input :

```
cholesky([[1,-1],[-1,4]])
```

or :

```
cholesky([[1,-3],[1,4]])
```

Output :

```
[[1,0],[-1,sqrt(3)]]
```

### 2.48.2 QR decomposition : `qr`

`qr` takes as argument a numeric square matrix $A$ of size $n$.
`qr` factorizes numerically this matrix as $Q * R$ where $Q$ is an orthogonal matrix ($^tQ * Q = I$) and $R$ is an upper triangular matrix. `qr(A)` returns only `R`, run `Q=A*inv(R)` to get `Q`.
Input :

```
qr([[3,5],[4,5]])
```

Output is the matrix `R` :

```
[[-5,-7],[0,-1]]
```

Input :

```
qr([[1,2],[3,4]])
```

Output is the matrix `R` :

```
[[-3.16227766017,-4.42718872424],[0,-0.632455532034]]
```

### 2.48.3 QR decomposition (for TI compatibility) : `QR`

`QR` takes as argument a numeric square matrix $A$ of size $n$ and two variable names, `var1` and `var2`.
`QR` factorizes this matrix numerically as $Q * R$ where $Q$ is an orthogonal matrix ($^tQ * Q = I$) and $R$ is an upper triangular matrix. `QR(A,var1,var2)` returns `R`, stores `Q=A*inv(R)` in `var1` and `R` in `var2`.
Input :

```
QR([[3,5],[4,5]],Q,R)
```

Output the matrix `R` :

```
[[-5,-7],[0,-1]]
```

Then input :

```
Q
```

Output the matrix `Q` :

```
[[-0.6,-0.8],[-0.8,0.6]]
```

### 2.48.4 LU decomposition : `lu`

`lu` takes as argument a square matrix $A$ of size $n$ (numeric or symbolic).
`lu(A)` returns a permutation $p$ of $0..n-1$, a lower triangular matrix $L$, with 1 on the diagonal, and an upper triangular matrix $U$, such that :

- $P * A = L * U$ where $P$ is the permutation matrix associated to $p$ (that may be computed by `P:=permu2mat(p)`),

- the equation $A * x = B$ is equivalent to :

$$L{*}U{*}x = P{*}B = p(B) \text{ where } p(B) = [b_{p(0)}, b_{p(1)}..b_{p(n-1)}], \quad B = [b_0, b_1..b_{n-1}]$$

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 $p$. The function `permu2mat` may be used to compute $P$ (`permu2mat(p)` returns P).
Input :

```
(p,L,U):=lu([[3.,5.],[4.,5.]])
```

Output :

```
[1,0],[[1,0],[0.75,1]],[[4,5],[0,1.25]]
```

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:

```
[[4.0,5.0],[3.0,5.0]],[[4.0,5.0],[3.0,5.0]]
```

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 :

```
[1,0],[[1,0],[3,1]],[[1,2],[0,-2]]
```

Input :

```
lu([[1.0,2],[3,4]])
```

Output :

```
[1,0],[[1,0],[0.333333333333,1]],[[3,4],
        [0,0.666666666667]]
```

### 2.48.5 LU decomposition (for TI compatibility) : `LU`

`LU` takes as argument a numeric square matrix $A$ of size $n$ and three variable names, `var1`, `var2` and `var3`.
`LU(A,var1,var2,var3)` returns $P$, a permutation matrix, and stores :

- a lower triangular matrix $L$, with 1 on the diagonal, in `var1`,
- an upper triangular matrix $U$ in `var2`,
- the permutation matrix $P$, result of the command `LU`, in `var3`.

These matrix are such that

the equation $A * x = B$ is equivalent to $L * U * x = P * B$.

Input :

```
LU([[3,5],[4,5]],L,U,P)
```

Output :

```
[[0,1],[1,0]]
```

Input :

```
L
```

Output :

```
[[1,0],[0.75,1]]
```

Input :

```
U
```

Output :

```
[[4,5],[0,1.25]]
```

Input :

```
P
```

Output :

```
[[0,1],[1,0]]
```

### 2.48.6 Singular value decomposition : `svd`

`svd` (singular value decomposition) takes as argument a numeric square matrix of size $n$.
`svd(A)` returns an orthogonal matrix $U$, the diagonal $s$ of a diagonal matrix $S$ and an orthogonal matrix $Q$ (${}^tQ * Q = I$) such that :

$$A = U.S.{}^tQ$$

Input :

```
svd([[1,2],[3,4]])
```

Output :

```
[[-0.404553584834,-0.914514295677],[-0.914514295677,
  0.404553584834]], [5.46498570422,0.365966190626],
  [[-0.576048436766,0.81741556047],[-0.81741556047,
                  -0.576048436766]]
```

Input :

```
(U,s,Q):=svd([[3,5],[4,5]])
```

Output :

```
[[-0.672988041811,-0.739653361771],[-0.739653361771,
   0.672988041811]],[8.6409011028,0.578643354497],
  [[-0.576048436766,0.81741556047],[-0.81741556047,
                  -0.576048436766]]
```

Verification : Input :

```
U*diag(s)*tran(Q)
```

Output :

```
[[3.0,5.0],[4.0,5.0]]
```

### 2.48.7 Short basis of a lattice : `lll`

`lll` takes as argument an invertible matrix $M$ with integer coefficients.
`lll` returns $(S, A, L, O)$ such that:

- 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 $M$ ($A * M = S$),
- $L$ is a lower triangular matrix, the modulus of it's non diagonal coefficients are less than 1/2,
- $O$ is a matrix with orthogonal rows such that $L * O = S$.

Input :

```
(S,A,L,O):=lll(M:=[[2,1],[1,2]])
```

Output :

```
[[-1,1],[2,1]], [[-1,1],[1,0]], [[1,0],[1/-2,1]],
              [[-1,1],[3/2,3/2]]
```

Hence :
`S=[[-1,1],[2,1]]`
`A=[[-1,1],[1,0]]`
`L=[[1,0],[1/-2,1]]`
`O=[[-1,1],[3/2,3/2]]`
Hence the original basis is `v1=[2,1], v2=[1,2]`
and the short basis is `w1=[-1,1], w2=[2,1]`.
Since `w1=-v1+v2` and `w2=v1` then :
`A:=[[-1,1],[1,0]]`, `A*M==S` and `L*O==S`.
Input :

```
(S,A,L,O):=lll([[3,2,1],[1,2,3],[2,3,1]])
```

Output :

```
S=[[-1,1,0],[-1,-1,2],[3,2,1]]

A= [[-1,0,1],[0,1,-1],[1,0,0]]

L= [[1,0,0],[0,1,0],[(-1)/2,(-1)/2,1]]

O= [[-1,1,0],[-1,-1,2],[2,2,2]]
```

Input :
`M:=[[3,2,1],[1,2,3],[2,3,1]]`
Properties :
`A*M==S` and `L*O==S`

## 2.49 Quadratic forms

### 2.49.1 Matrix of a quadratic form : `q2a`

`q2a` takes two arguments : the symbolic expression of a quadratic form $q$ and a vector of variable names.
`q2a` returns the matrix $A$ of $q$.
Input :

```
q2a(2*x*y,[x,y])
```

Output :

```
[[0,1],[1,0]]
```

### 2.49.2 Transform a matrix into a quadratic form : `a2q`

`a2q` takes two arguments : the symetric matrix $A$ of a quadratic form $q$ and a vector of variable names of the same size.
`a2q` returns the symbolic expression of quadratic form $q$.
Input :

```
a2q([[0,1],[1,0]],[x,y])
```

Output :

```
2*x*y
```

Input :

```
a2q([[1,2],[2,4]],[x,y])
```

Output :

```
x^2+4*x*y+4*y^2
```

### 2.49.3 Reduction of a quadratic form : `gauss`

`gauss` takes two arguments : a symbolic expression representing a quadratic form $q$ and a vector of variable names.
`gauss` returns $q$ written as sum or difference of squares using Gauss algorithm.
Input :

```
gauss(2*x*y,[x,y])
```

Output :

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

### 2.49.4 Gramschmidt orthonormalization : `gramschmidt`

`gramschmidt` takes one or two arguments :

- a matrix viewed as a list of row vectors, the scalar product being the canonical scalar product, or
- a list of elements that is a basis of a vector subspace, and a function that defines a scalar product on this vector space.

`gramschmidt` returns an orthonormal basis for this scalar product.
Input :

```
normal(gramschmidt([[1,1,1],[0,0,1],[0,1,0]]))
```

Or input :

```
normal(gramschmidt([[1,1,1],[0,0,1],[0,1,0]],dot))
```

Output :

```
[[(sqrt(3))/3,(sqrt(3))/3,(sqrt(3))/3],[(-(sqrt(6)))/6,(-(sqrt(6)))/6,
```

**Example**
We define a scalar product on the vector space of polynomials by:

$$P.Q = \int_{-1}^{1} P(x).Q(x)dx$$

Input :

```
gramschmidt([1,1+x],(p,q)->integrate(p*q,x,-1,1))
```

Or define the function `p_scal`, input :
`p_scal(p,q):=integrate(p*q,x,-1,1)`
then input :

```
gramschmidt([1,1+x],p_scal)
```

Output :

```
[1/(sqrt(2)),(1+x-1)/sqrt(2/3)]
```

### 2.49.5 Graph of a conic : `conique`

`conique` takes as argument the equation of a conic with respect to $x, y$. You may also specify the names of the variables as second and third argument or as a vector as second argument.
`conique` draws this conic.
Input :

```
conique(2*x^2+2*x*y+2*y^2+6*x)
```

Output :

```
the graph of the ellipsis of center -2+i and equation
              2*x^2+2*x*y+2*y^2+6*x=0
```

**Remark** :
See also `conique_reduite` for the parametric equation of the conic.

### 2.49.6 Conic reduction : `conique_reduite`

`conique_reduite` takes two arguments : the equation of a conic and a vector of variable names.
`conique_reduite` 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 degenerated),
- the reduced equation of the conic

- a vector of it's parametric equations.

Input :

```
    conique_reduite(2*x^2+2*x*y+2*y^2+5*x+3,[x,y])
```

Output :

```
[[-5/3,5/6],[[-1/(sqrt(2)),1/(sqrt(2))],[-1/(sqrt(2)),
    -1/(sqrt(2))]],1,3*x^2+y^2+-7/6,[[(-10+5*i)/6+
  (1/(sqrt(2))+(i)/(sqrt(2)))*((sqrt(14)*cos(` t`))/6+
  ((i)*sqrt(42)*sin(` t`))/6),` t`,0,2*pi,(2*pi)/60]]]
```

Which means that the conic is not degenerated, it's reduced equation is

$$3x^2 + y^2 - 7/6 = 0$$

origin is $-5/3 + 5 * i/6$, axis are parallel to the vectors $(-1, 1)$ and $(-1, -1)$. It's parametric equation is

$$\frac{-10 + 5 * i}{6} + \frac{(1 + i)}{\sqrt{2}} * \frac{(\sqrt{14} * cos(t) + i * \sqrt{42} * sin(t))}{6}$$

where the suggested parameters value for drawing are $t$ from 0 to $2\pi$ with `tstep`= $2\pi/60$.

**Remark** :
Note that if the conic is degenerated 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.
Input :

```
conique_reduite(x^2-y^2+3*x+y+2)
```

Output :

```
[[(-3)/2,1/2],[[1,0],[0,1]],0,x^;2-y^2,
    [[(-1+2*i)/(1-i),(1+2*i)/(1-i)],
      [(-1+2*i)/(1-i),(-1)/(1-i)]]]
```

### 2.49.7 Graph of a quadric : `quadrique`

`quadrique` takes as arguments the expression of a quadric with respect to $x, y, z$. You may also specify the variables as a vector (second argument) or as second, third and fourth arguments.
`quadrique` draws this quadric.
Input :

```
quadrique(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 :

```
        the drawing of the ellipsoid of equation
 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=0
```

See also `quadrique_reduite` for the parametric equation of the quadric.

### 2.49.8 Quadric reduction : `quadrique_reduite`

`quadrique_reduite` takes two arguments : the equation of a quadric and a vector of variable names.
`quadrique_reduite` 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 degenerated),
- 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 `quadrique_reduite`).
Input :

```
quadrique_reduite(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 is a list containing :

- The origin (center of symmetry) of the quadric

```
[11/27,(-26)/27,(-29)/54],
```

- The matrix of the basis change:

```
[[(sqrt(6))/3,(sqrt(5))/5,(-(sqrt(30)))/15],
          [(sqrt(6))/6,0,(sqrt(30))/6],
 [(-(sqrt(6)))/6,(2*sqrt(5))/5,(sqrt(30))/30]],
```

- 1 hence the quadric is not degenerated
- the reduced equation of the quadric :

```
0,9*x^2+3*y^2+3*z^2+(-602)/27,
```

- The parametric equations (in the original frame) are :

```
[[(sqrt(6)*sqrt(602/243)*sin(u)*cos(v))/3+
  (sqrt(5)*sqrt(602/81)*sin(u)*sin(v))/5+
((-(sqrt(30)))*sqrt(602/81)*cos(u))/15+11/27,
  (sqrt(6)*sqrt(602/243)*sin(u)*cos(v))/6+
 (sqrt(30)*sqrt(602/81)*cos(u))/6+(-26)/27,
((-(sqrt(6)))*sqrt(602/243)*sin(u)*cos(v))/6+
 (2*sqrt(5)*sqrt(602/81)*sin(u)*sin(v))/5+
(sqrt(30)*sqrt(602/81)*cos(u))/30+(-29)/54], u=(0
  ..  pi),v=(0..  (2*pi)),ustep=(pi/20),
          vstep=((2*pi)/20)]]
```

Hence the quadric is an ellipsoid and its reduced equation is :

$$9 * x^2 + 3 * y^2 + 3 * z^2 + (-602)/27$$

after the change of origin $[11/27, (-26)/27, (-29)/54]$, the matrix of basis change `P` 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 degenerated and made of 1 or 2 plan(s), each plan is not given by its parametric equation but by the list of a point of the plan and of a normal vector to the plan.
Input :

```
quadrique_reduite(x^2-y^2+3*x+y+2)
```

Output :

```
[[(-3)/2,1/2,0],[[1,0,0],[0,1,0],[0,0,-1]],0,x^2-y^2,
        [hyperplan([1,1,0],[(-3)/2,1/2,0]),
       hyperplan([1,-1,0],[(-3)/2,1/2,0])]]
```

## 2.50 Multivariate calculus

### 2.50.1 Gradient : `derive deriver diff grad`

`derive` (or `diff` or `grad`) takes two arguments : an expression $F$ of $n$ real variables and a vector of these variable names.
`derive` returns the gradient of $F$, where the gradient is the vector of all partial derivatives, for exmple in dimension $n = 3$

$$\overrightarrow{\text{grad}}(F) = [\frac{\partial F}{\partial x}, \frac{\partial F}{\partial y}, \frac{\partial F}{\partial z}]$$

**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*2*x*y-z^3,2*x^2,-(x*3*z^2)]
```

Output after simplification with `normal(ans())` :

```
[4*x*y-z^3,2*x^2,-(3*x*z^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]]
```

### 2.50.2 Laplacian : `laplacian`

`laplacian` takes two arguments : an expression $F$ of $n$ real variables and a vector of these variable names.
`laplacian` returns the laplacian de $F$, that is the sum of all second partial derivatives, for example in dimension $n = 3$:

$$\nabla^2(F) = \frac{\partial^2 F}{\partial x^2} + \frac{\partial^2 F}{\partial y^2} + \frac{\partial^2 F}{\partial z^2}$$

**Example**
Find the laplacien of $F(x, y, z) = 2x^2y - xz^3$.
Input :

```
laplacian(2*x^2*y-x*z^3,[x,y,z])
```

Output :

```
4*y+-6*x*z
```

### 2.50.3 Hessian matrix : `hessian`

`hessian` takes two arguments : an expression $F$ of $n$ real variables and a vector of these variable names.
`hessian` returns the hessian matrix of $F$, that is the matrix of the derivatives of order 2.
**Example**
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 :

```
[[4*y,4*x,-(3*z^2)],[2*2*x,0,0],[-(3*z^2),0,x*3*2*z]]
```

To have the hessian matrix at the critical points, first input :

```
solve(derive(2*x^2*y-x*z^3,[x,y,z]),[x,y,z])
```

Output is the critical points :

```
[[0,y,0]]
```

Then, to have the hessian matrix at this points, input :

```
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 :

```
[[4*y,4*0,-(3*0^2)],[4*0,0,0],[-(3*0^2),0,6*0*0]]
```

and after simplification :

```
[[4*y,0,0],[0,0,0],[0,0,0]]
```

### 2.50.4 Divergence : `divergence`

`divergence` takes two arguments : a vector field of dimension $n$ depending on $n$ real variables.
`divergence` returns the divergence of $F$ that is the sum of the derivative of the $k$-th component with respect to the $k$-th variable. For example in dimension $n = 3$:

$$\texttt{divergence([A,B,C],[x,y,z])}=\frac{\partial A}{\partial x}+\frac{\partial B}{\partial y}+\frac{\partial C}{\partial z}$$

Input :

```
divergence([x*z,-y^2,2*x^y],[x,y,z])
```

Output :

```
z+-2*y
```

### 2.50.5 Rotationnal : `curl`

`curl` takes two arguments : a 3-d vector field depending on 3 variables.
`curl` returns the rotationnal of the vector, defined by:

$$\texttt{curl([A,B,C],[x,y,z])}=[\frac{\partial C}{\partial y}-\frac{\partial B}{\partial z},\frac{\partial A}{\partial z}-\frac{\partial C}{\partial x},\frac{\partial B}{\partial x}-\frac{\partial A}{\partial y}]$$

Note that $n$ **must be equal to 3**.
Input :

```
curl([x*z,-y^2,2*x^y],[x,y,z])
```

Output :

```
[2*x^y*log(x),x-2*y*x^(y-1),0]
```

### 2.50.6 Potential : potential

potential takes two arguments : a vector field $\overrightarrow{V}$ in $R^n$ with respect to $n$ real variables and the vector of these variable names.
potential returns, if it is possible, a function $U$ such that $\overrightarrow{\text{grad}}(U) = \overrightarrow{V}$. When it is possible we say that $\overrightarrow{V}$ derive of the potential $U$, and $U$ is defined up to a constant.
potential is the reciprocal function of derive.
Input :

```
potential([2*x*y+3,x^2-4*z,-4*y],[x,y,z])
```

Output :

```
2*y*x^2/ 2+3*x+(x^2-4*z-2*x^2/2)*y
```

Note that in $\mathbb{R}^3$ a vector $\overrightarrow{V}$ is a gradient if and only if it's rotationnal is zero i.e. if curl(V)=0. In time-independant electro-magnetism, $\overrightarrow{V}$=$\overrightarrow{E}$ is the electric field and $U$ is the electric potential.

### 2.50.7 Conservative flux field : vpotential

vpotential takes two arguments : a vector field $\overrightarrow{V}$ in $R^n$ with respect to $n$ real variables and the vector of these variable names.
vpotential returns, if it is possible, a vector $\overrightarrow{U}$ such that $\overrightarrow{\text{curl}}(\overrightarrow{U}) = \overrightarrow{V}$. When it is possible we say that $\overrightarrow{V}$ is a conservative flux field or a solenoidal field. The general solution is the sum of a particular solution and of the gradient of an arbitrary function, Xcas returns a particular solution with zero as first component.
vpotential is the reciprocal function of curl.
Input :

```
vpotential([2*x*y+3,x^2-4*z,-2*y*z],[x,y,z])
```

Output :

```
[0,(-(2*y))*z*x,-x^3/3-(-(4*z))*x+3*y]
```

In $\mathbb{R}^3$, a vector field $\overrightarrow{V}$ is a rotationnal if and only if it's divergence is zero (divergence(V,[x,y,z])=0). In time-independant electro-magnetism, $\overrightarrow{V}$= $\overrightarrow{B}$ is the magnetic field and $\overrightarrow{U}$= $\overrightarrow{A}$ is the potential vector.

## 2.51 Equations

### 2.51.1 Define an equation : equal

equal takes as argument the two member of an equation.
equal returns this equation. It is the prefixed version of =
Input :

```
equal(2x-1,3)
```

Output :

```
(2*x-1)=3
```

We can also directly write (2*x-1)=3.

### 2.51.2 Transform an equation into a difference : `equal2diff`

`equal2diff` takes as argument an equation.
`equal2diff` returns the difference of the two members of this equation.
Input :

```
equal2diff(2x-1=3)
```

Output :

```
2*x-1-3
```

### 2.51.3 Transform an equation into a list : `equal2list`

`equal2list` takes as argument an equation.
`equal2list` returns the list of the two members of this equation.
Input :

```
equal2list(2x-1=3)
```

Output :

```
[2*x-1,3]
```

### 2.51.4 The left member of an equation : `left gauche lhs`

`left` or `lhs` takes as argument an equation or an interval.
`left` or `lhs` returns the left member of this equation or the left bound of this interval.
Input :

```
left(2x-1=3)
```

Or input:

```
lhs(2x-1=3)
```

Output :

```
2*x-1
```

Input :

```
left(1..3)
```

Or input:

```
lhs(1..3)
```

Output :

```
1
```

### 2.51.5 The right member of an equation : `right droit rhs`

`right` or `rhs` takes as argument an quation or an interval.
`right` or `rhs` returns the right member of this equation or the right bound of this interval.
Input :

```
right(2x-1=3)
```

Or :

```
rhs(2x-1=3)
```

Output :

```
3
```

Input :

```
right(1..3)
```

Or :

```
rhs(1..3)
```

Output :

```
3
```

### 2.51.6 Solving equation(s): `solve`

`solve` solves an equation or a system of polynomial equations. It takes 2 arguments:

- Solving an equation
  `solve` takes as arguments an equation between two expressions or an expression (`=0` is omitted), and a variable name (by default `x`).
  `solve` solves this equation.

- Solving a system of polynomial equations
  `solve` takes as arguments two vectors : a vector of polynomial equations and a vector of variable names.
  `solve` solves this polynomial equation system.

**Remarks**:

- In real mode, `solve` returns only real solutions. To have the complex solutions, switch to complex mode, e.g. by checking `Complex` in the cas configuration, or use the `cSolve` command.

- For trigonometric equations, `solve` returns by default the principal solutions. To have all the solutions check `All_trig_sol` in the cas configuration.

**Examples** :

- Solve $x^4 - 1 = 3$
  Input :

  ```
  solve(x^4-1=3)
  ```

  Output in real mode :

  ```
  [sqrt(2),-(sqrt(2))]
  ```

  Output in complex mode :

  ```
  [sqrt(2),-(sqrt(2)),(i)*sqrt(2),-((i)*sqrt(2))]
  ```

- Solve $\exp(x) = 2$
  Input :

  ```
  solve(exp(x)=2)
  ```

  Output in real mode :

  ```
  [log(2)]
  ```

- Find $x, y$ such that $x + y = 1, x - y = 0$
  Input :

  ```
  solve([x+y=1,x-y],[x,y])
  ```

  Output :

  ```
  [[1/2,1/2]]
  ```

- 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 :

  ```
  [[-2,-2],[1,1],[(-sqrt(5)+1)/2,(1+sqrt(5))/2],

  [(sqrt(5)+1)/2,(1-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]]
```

- Solve $\cos(2*x) = 1/2$
  Input :

  ```
  solve(cos(2*x)=1/2)
  ```

  Output :

  ```
  [pi/6,(-pi)/6]
  ```

  Output with `All_trig_sol` checked :

  ```
  [(6*pi*n_0+pi)/6,(6*pi*n_0-pi)/6]
  ```

- Find the intersection of a straight line (given by a list of equations) and a plane.
  For example, let $D$ be the straight line of cartesian equations $[y - z = 0, z - x = 0]$ and let $P$ the plane of equation $x-1+y+z = 0$. Find the intersection of $D$ and $P$.
  Input :

  ```
  solve([[y-z=0,z-x=0],x-1+y+z=0],[x,y,z])
  ```

  Output :

  ```
  [[1/3,1/3,1/3]]
  ```

### 2.51.7 Equation solving in $\mathbb{C}$ : `cSolve`

`cSolve` takes two arguments and solves an equation or a system of polynomial equations.

- solving an equaion
  `cSolve` takes as arguments an equation between two expressions or an expression (=`0` is omitted), and a variable name (by default `x`).
  `cSolve` solves this equation in $\mathbb{C}$ even if you are in real mode.
- solving system of polynomial equations
  `cSolve` takes as arguments two vectors : a vector of polynomial equations and a vector of variable names.
  `cSolve` solves this equation system in $\mathbb{C}$ even if you are in real mode.

Input :

```
cSolve(x^4-1=3)
```

Output :

```
[sqrt(2),-(sqrt(2)),(i)*sqrt(2),-((i)*sqrt(2))]
```

Input :

```
cSolve([-x^2+y=2,x^2+y],[x,y])
```

Output :

```
[[i,1],[-i,1]]
```

## 2.52 Linear systems

In this paragraph, we call "augmented matrix" of the system $A \cdot X = B$ (or matrix "representing" the system $A \cdot X = B$), the matrix obtained by gluing the column vector $B$ or $-B$ to the right of the matrix $A$, as with `border(A,tran(B))`.

### 2.52.1 Matrix of a system : `syst2mat`

`syst2mat` takes two vector as argument. The components of the first vector are the equations of a linear system and the components of the second vector are the variable names.
`syst2mat` returns the augmented matrix of the system $AX = B$, obtained by gluing the column vector $-B$ to the right of the matrix $A$.
Input :

```
syst2mat([x+y,x-y-2],[x,y])
```

Output :

```
[[1,1,0],[1,-1,-2]]
```

Input :

```
syst2mat([x+y=0,x-y=2],[x,y])
```

Output :

```
[[1,1,0],[1,-1,-2]]
```

**Warning !!!**
The variables (here `x` and `y`) must be purged.

### 2.52.2 Gauss reduction of a matrix : `ref`

`ref` is used to solve a linear system of equations written in matrix form:

```
A*X=B
```

The argument of `ref` is the augmented matrix of the system (the matrix obtained by augmenting the matrix `A` to the right with the column vector `B`).
The result is a matrix `[A1,B1]` where `A1` has zeros under its principal diagonal, and the solutions of:

```
A1*X=B1
```

are the same as the solutions of:

```
A*X=B
```

For example, solve the system :

$$\left\{ \begin{array}{lcr} 3x+y & = & -2 \\ 3x+2y & = & 2 \end{array} \right.$$

Input :

```
ref([[3,1,-2],[3,2,2]])
```

Output :

```
[[1,1/3,-2/3],[0,1,4]]
```

Hence the solution is $y = 4$ (last row) and $x = -2$ (substitute $y$ in the first row).

### 2.52.3 Gauss-Jordan reduction: `rref gaussjord`

`rref` solves a linear system of equations written in matrix form (see also 2.31.17) :

```
A*X=B
```

`rref` takes one or two arguments.

- If `rref` has only one argument, this argument is the augmented matrix of the system (the matrix obtained by augmenting matrix `A` to the right with the column vector `B`).
  The result is a matrix `[A1,B1]` : `A1` has zeros both above and under its principal diagonal and has 1 on its principal diagonal, and the solutions of:

  ```
  A1*X=B1
  ```

  are the same as :

  ```
  A*X=B
  ```

  For example, to solve the system:

  $$\left\{\begin{array}{rcr} 3x+y & = & -2 \\ 3x+2y & = & 2 \end{array}\right.$$

  input :

  ```
  rref([[3,1,-2],[3,2,2]])
  ```

  Output :

  ```
  [[1,0,-2],[0,1,4]]
  ```

  Hence $x = -2$ and $y = 4$ is the solution of this system.

  `rref` can also solve several linear systems of equations having the same first member. We write the second members as a column matrix.
  Input :

  ```
  rref([[3,1,-2,1],[3,2,2,2]])
  ```

  Output :

  ```
  [[1,0,-2,0],[0,1,4,1]]
  ```

  Which means that ($x = -2$ and $y = 4$) is the solution of the system

  $$\left\{\begin{array}{rcr} 3x+y & = & -2 \\ 3x+2y & = & 2 \end{array}\right.$$

  and ($x = 0$ and $y = 1$) is the solution of the system

  $$\left\{\begin{array}{rcl} 3x+y & = & 1 \\ 3x+2y & = & 2 \end{array}\right.$$

- If `rref` has two parameters, the second parameter must be an integer $k$, and the Gauss-Jordan reduction will be performed on (at most) the first $k$ columns.
  Input :

  ```
  rref([[3,1,-2,1],[3,2,2,2]],1)
  ```

  Output :

  ```
  [[3,1,-2,1],[0,1,4,1]]
  ```

### 2.52.4 Solving A*X=B : `simult`

`simult` is used to solve a linear system of equations (resp. several linear systems of equations with the same matrix `A`) written in matrix form (see also 2.31.17) :

```
A*X=b (resp A*X=B)
```

`simult` takes as arguments the matrix `A` of the system and the column vector (i.e. a one column matrix) `b` of the second member of the system (resp. the matrix `B` whose columns are the vectors `b` of the second members of the different systems). The result is a column vector, solution of the system (resp. a matrix whose columns are the solutions of the different systems).
For example, to solve the system :

$$\left\{\begin{array}{rcr} 3x+y & = & -2 \\ 3x+2y & = & 2 \end{array}\right.$$

input :

```
simult([[3,1],[3,2]],[[-2],[2]])
```

Output :

```
[[-2],[4]]
```

Hence $x=-2$ and $y=4$ is the solution.
Input :

```
simult([[3,1],[3,2]],[[-2,1],[2,2]])
```

Output :

```
[[-2,0],[4,1]]
```

Hence $x=-2$ and $y=4$ is the solution of

$$\left\{\begin{array}{rcr} 3x+y & = & -2 \\ 3x+2y & = & 2 \end{array}\right.$$

whereas $x=0$ and $y=1$ is the solution of

$$\left\{\begin{array}{rcr} 3x+y & = & 1 \\ 3x+2y & = & 2 \end{array}\right.$$

### 2.52.5 Step by step Gauss-Jordan reduction of a matrix : `pivot`

`pivot` takes three arguments : a matrix with $n$ rows and $p$ columns and two integers $l$ and $c$ such that $0 \leq l < n$, $0 \leq c < p$ and $A_{l,c} \neq 0$.
`pivot(A,l,c)` performs one step of the Gauss-Jordan method using `A[l,c]` as pivot and returns an equivalent matrix with zeros in the column `c` of `A` (except at row $l$).
Input :

```
pivot([[1,2],[3,4],[5,6]],1,1)
```

Output :

```
[[-2,0],[3,4],[2,0]]
```

Input :

```
pivot([[1,2],[3,4],[5,6]],0,1)
```

Output :

```
[[1,2],[2,0],[4,0]]
```

### 2.52.6 Linear system solving: `linsolve`

`linsolve` is used to solve a system of linear equations.
`linsolve` has two arguments: a list of equations or expressions (in that case the convention is that the equation is $expression = 0$), and a list of variable names.
`linsolve` returns the solution of the system in a list.
Input :

```
linsolve([2*x+y+z=1,x+y+2*z=1,x+2*y+z=4],[x,y,z])
```

Output :

```
[1/-2,5/2,1/-2]
```

Which means that

$$x = -\frac{1}{2}, y = \frac{5}{2}, z = -\frac{1}{2}$$

is the solution of the system :

$$\begin{cases} 2x + y + z & = 1 \\ x + y + 2z & = 1 \\ x + 2y + z & = 4 \end{cases}$$

### 2.52.7 Finding linear recurrences : `reverse_rsolve`

`reverse_rsolve` takes as argument a vector $v = [v_0 ... v_{2n-1}]$ made of the first $2n$ terms of a sequence $(v_n)$ which is supposed to verify a linear recurrence relation of degree smaller than $n$

$$x_n * v_{n+k} + ... + x_0 * v_k = 0$$

where the $x_j$ are $n+1$ unknowns.
`reverse_rsolve` returns the list $x=[x_n,...,x_0]$ of the $x_j$ coefficients (if $x_n \neq 0$ it is reduced to 1).

In other words `reverse_rsolve` solves the linear system of $n$ equations :

$$\begin{aligned} x_n * v_n + ... + x_0 * v_0 &= 0 \\ &... \\ x_n * v_{n+k} + ... + x_0 * v_k &= 0 \\ &... \\ x_n * v_{2*n-1} + ... + x_0 * v_{n-1} &= 0 \end{aligned}$$

The matrix $A$ of the system has $n$ rows and $n+1$ columns :

$$A = [[v_0, v_1...v_n], [v_1, v_2, ...v_{n-1}], ..., [v_{n-1}, v_n...v_{2n-1}]]$$

`reverse_rsolve` returns the list $x=[x_n,...x_1,x_0]$ with $x_n=1$ and $x$ is the solution of the system $A*\texttt{revlist}(x)$.

**Examples**

- Find a sequence verifying a linear recurrence of degree at most 2 whose first elements 1, -1, 3, 3.
  Input :

  ```
  reverse_rsolve([1,-1,3,3])
  ```

  Output :

  ```
  [1,-3,-6]
  ```

  Hence $x_0=-6$, $x_1=-3$, $x_2=1$ and the recurrence relation is

  $$v_{k+2} - 3v_{k+1} - 6v_k = 0$$

  Without `reverse_rsolve`, we would write the matrix of the system :
  `[[1,-1,3],[-1,3,3]]` and use the `rref` command :
  `rref([[1,-1,3],[-1,3,3]])`
  Output is `[[1,0,6],[0,1,3]]` hence $x_0=-6$ and $x_1=-3$ (because $x_2=1$).

- Find a sequence verifying a linear recurrence of degree at most 3 whose first elements are 1, -1, 3, 3,-1, 1.
  Input :

  ```
  reverse_rsolve([1,-1,3,3,-1,1])
  ```

  Output :

  ```
  [1,(-1)/2,1/2,-1]
  ```

Hence so, $x_0 = -1$, $x_1 = 1/2$, $x_2 = -1/2$, $x_3 = 1$, the recurrence relation is

$$v_{k+3} - \frac{1}{2}v_{k+2} + \frac{1}{2}v_{k+1} - v_k = 0$$

Without `reverse_rsolve`, we would write the matrix of the system :
`[[1,-1,3,3],[-1,3,3,-1],[3,3,-1,1]]`.
Using `rref` command, we would input :
`rref([[1,-1,3,3],[-1,3,3,-1],[3,3,-1,1]])`
Output is `[1,0,0,1],[0,1,0,1/-2],[0,0,1,1/2]]` hence $x_0 = -1$, $x_1 = 1/2$ and $x_2 = -1/2$ because $x_3 = 1$),

## 2.53 Differential equations

This section is limited to symbolic (or exact) solutions of differential equations. For numeric solutions of differential equations, see `odesolve`. For graphic representation of solutions of differential equations, see `plotfield`, `plotode` and `interactive_plotode`.

### 2.53.1 Solving differential equations : `desolve deSolve dsolve`

`desolve` (or `deSolve`) 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 separated 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` takes as arguments :

- if the independant variable is the current variable (here supposed to be $x$),
  - the differential equation (or the list of the differential equation and of the initial conditions)
  - the unknown (usually `y`).

  In the differential equation, the function $y$ is denoted by $y$, it's first derivative $y\prime$ is denoted by y$'$, and it's second derivative $y''$ is written y$''$.
  For example `desolve(y''+2*y'+y,y)` or
  `desolve([y''+2*y'+y,y(0)=1,y'(0)=0],y)`.

- if the independant variable is not the current variable, for example $t$ instead of $x$,
  - the differential equation (or the list of the differential equation and of the initial conditions),
  - the variable, e.g. `t`
  - the unknown as a variable `y` or as a function `y(t)`.

  In the differential equation, the function $y$ is denoted by $y(t)$, it's derivative $y\prime$ is denoted by `diff(y(t),t)`, and it's second derivative $y''$ is denoted by `diff(y(t),t$2)`.
  For example :
  `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);` and

```
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);
```

If there is no initial conditions (or one initial condition for a second order equation), `desolve` returns the general solution in terms of constants of integration `c_0`, `c_1`, where `y(0)=c_0` and `y'(0)=c_1`, or a list of solutions.
**Examples**

- Examples of second linear differential equations with constant coefficients.

  1. Solve :
     $$y'' + y = \cos(x)$$
     Input (typing twice prime for `y''`):
     ```
     desolve(y''+y=cos(x),y)
     ```
     or input :
     ```
     desolve((diff(diff(y))+y)=(cos(x)),y)
     ```
     Output :
     ```
     c_0*cos(x)+(x+2*c_1)*sin(x)/2
     ```
     `c_0, c_1` are the constants of integration : `y(0)=c_0` and `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)+(t+2*c_1)/2*sin(t)
     ```
     `c_0, c_1` are the constants of integration : `y(0)=c_0` and `y'(0)=c_1`.

2. Solve :
$$y'' + y = \cos(x), \quad y(0) = 1$$
Input :
```
desolve([y''+y=cos(x),y(0)=1],y)
```
Output :
```
[cos(x)+(x+2*c_1)/2*sin(x)]
```
the components of this vector are solutions (here there is just one component, so we have just one solution depending of the constant `c_1`).

3. Solve :
$$y'' + y = \cos(x) \quad (y(0))^2 = 1$$
Input :
```
desolve([y''+y=cos(x),y(0)^2=1],y)
```
Output :
```
[-cos(x)+(x+2*c_1)/2*sin(x),cos(x)+(x+2*c_1)/2*sin(x)]
```
each component of this list is a solution, we have two solutions depending on the constant `c_1` ($y'(0) = c_1$) and corresponding to $y(0) = 1$ and to $y(0) = -1$.

4. Solve :
$$y'' + y = \cos(x), \quad (y(0))^2 = 1 \;\; y'(0) = 1$$
Input :
```
desolve([y''+y=cos(x),y(0)^2=1,y'(0)=1],y)
```
Output :
```
[-cos(x)+(x+2)/2*sin(x),cos(x)+(x+2)/2*sin(x)]
```
each component of this list is a solutions (we have two solutions).

5. Solve :
$$y'' + 2y' + y = 0$$
Input :
```
desolve(y''+2*y'+y=0,y)
```
Output :
```
(x*c_0+x*c_1+c_0)*exp(-x)
```
the solution depends of 2 constants of integration : `c_0, c_1` (`y(0)=c_0` and `y'(0)=c_1`).

6. Solve :
$$y'' - 6y' + 9y = xe^{3x}$$
Input:
```
desolve(y''-6*y'+9*y=(x*exp(3*x),y)
```

Output :

```
(x^3+(-(18*x))*c_0+6*x*c_1+6*c_0)*1/6*exp(3*x)
```

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 differential equations.

  1. Solve :
  $$xy' + y - 3x^2 = 0$$
  Input :
  ```
  desolve(x*y'+y-3*x^2,y)
  ```
  Output :
  ```
  (3*1/3*x^3+c_0)/x
  ```
  2. 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 :
  ```
  [1/(exp(1/2*x^2))]
  ```
  3. Solve :
  $$x(x^2 - 1)y' + 2y = 0$$
  Input :
  ```
  desolve(x*(x^2-1)*y'+2*y=0,y)
  ```
  Output :
  ```
  (c_0)/((x^2-1)/(x^2))
  ```
  4. Solve :
  $$x(x^2 - 1)y' + 2y = x^2$$
  Input :
  ```
  desolve(x*(x^2-1)*y'+2*y=x^2,y)
  ```
  Output :
  ```
  (ln(x)+c_0)/((x^2-1)/(x^2))
  ```
  5. If the variable is $t$ instead of $x$, for example :
  $$t(t^2 - 1)y'(t) + 2y(t) = t^2$$
  Input :

```
desolve(t*(t^2-1)*diff(y(t),t)+2*y(t)=(t^2),y(t))
```

Output :

```
(ln(t)+c_0)/((t^2-1)/(t^2))
```

6. Solve :

$$x(x^2-1)y'+2y=x^2, y(2)=0$$

Input :

```
desolve([x*(x^2-1)*y'+2*y=x^2,y(0)=1],y)
```

Output :

```
[(ln(x)-ln(2))*1/(x^2-1)*x^2]
```

7. 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 :

```
(-c_0+ln(sqrt(x^2+1)-x))/(x-sqrt(x^2+1))
```

- Examples of first differential equations with separated variables.

1. Solve :

$$y'=2\sqrt{y}$$

Input :

```
desolve(y'=2*sqrt(y),y)
```

Output :

```
[x^2+-2*x*c_0+c_0^2]
```

2. 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 $u=1/y^{n-1}$.

1. Solve :

$$xy'+2y+xy^2=0$$

Input :

```
desolve(x*y'+2*y+x*y^2,y)
```

Output :

```
[1/(exp(2*ln(x))*(-1/x+c_0))]
```

2. Solve :

$$xy' - 2y = xy^3$$

Input :

```
desolve(x*y'-2*y-x*y^3,y)
```

Output :

```
[((-2*1/5*x^5+c_0)*exp(-(4*log(x))))^(1/-2),
-((-2*1/5*x^5+c_0)*exp(-(4*log(x))))^(1/-2)]
```

3. Solve :

$$x^2y' - 2y = xe^(4/x)y^3$$

Input :

```
desolve(x*y'-2*y-x*exp(4/x)*y^3,y)
```

Output :

```
[((-2*ln(x)+c_0)*exp(-(4*(-(1/x)))))^(1/-2),
-(((-2*ln(x)+c_0)*exp(-(4*(-(1/x)))))^(1/-2))]
```

- Examples of first order homogeneous differential equations ($y' = F(y/x)$, the method of integration is to search $t = y/x$ instead of $y$).

1. Solve :

$$(3x^3y' = y(3x^2 - y^2)$$

Input :

```
desolve(3*x^3*diff(y)=((3*x^2-y^2)*y),y)
```

Output :

```
[0,pnt[c_0*exp((3*1/2)/(` t`^2)),`
  t`*c_0*exp((3*1/2)/(` t`^2))]]
```

hence the solutions are $y = 0$ and the familiy of curves of parametric equation $x = c_0 \exp(3/(2t^2)), y = t * c_0 \exp(3/(2t^2))$ (the parameter is denoted by `` ` t` `` in the answer).

2. Solve :

$$xy' = y + \sqrt{x^2 + y^2}$$

Input :

```
desolve(x*y'=y+sqrt(x^2+y^2),y)
```

Output :

```
[(-i)*x,(i)*x,pnt[c_0/(sqrt(` t`^2+1)-` t`),(`
     t`*c_0)/(sqrt(` t`^2+1)-` t`)]]
```

hence the solutions are :

$$y = ix, y = -ix$$

and the family of curves of parametric equations

$$x = c_0/(\sqrt{t^2+1} - t), y = t * c_0/(\sqrt{t^2+1} - t)$$

(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.

  1. Solve :
     $$yy' + x$$
     Input :
     ```
     desolve(y*y'+x,y)
     ```
     Output :
     ```
     [sqrt(-2*c_0-x^2),-(sqrt(-2*c_0-x^2))]
     ```
     In this example, $xdx + ydy$ is closed, the integrating factor was 1.
  2. Solve :
     $$2xyy' + x^2 - y^2 + a^2 = 0$$
     Input :
     ```
     desolve(2*x*y*y'+x^2-y^2+a^2,y)
     ```
     Output :
     ```
     [sqrt(a^2-x^2-c_1*x),-(sqrt(a^2-x^2-c_1*x))]
     ```
     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 equations can not be solved directly by `Xcas`, we explain how to solve them with it's help. 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, we put $y + y' = t$, 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 :

```
-11*1/9*ln(8*t^3+3)+1/-9*ln(t^3)+c_0
```

eventually 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.

  1. Solve :
     $$xy' + y'^3 - y) = 0$$
     Input :
     ```
     desolve(x*y'+y'^3-y),y)
     ```
     Output :
     ```
     c_0*x+c_0^3
     ```
  2. 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 :
     ```
     c_0*x+sqrt(a^2+b^2*c_0^2)
     ```

### 2.53.2 Laplace transform and inverse Laplace transform : `laplace` `ilaplace`

`laplace` and `ilaplace` takes one, two or three arguments : an expression and optionnaly the name(s) of the variable(s).
The expression is an expression of the current variable (here $x$) or an expression of the variable given as second argument.
`laplace` returns the Laplace transform of the expression given as argument and `ilaplace` the inverse Laplace transform of the expression given as argument. The result of `laplace` and `ilaplace` is expressed in terms of the variable given as third argument if supplied or second argument if supplied or $x$ otherwise.

Laplace transform (`laplace`) and inverse Laplace transform (`ilaplace`) are useful to solve linear differential equations with constant coefficients. For example :

$$y\prime\prime + p.y\prime + q.y \ = \ f(x)$$
$$y(0) = a, \ y\prime(0) = b$$

Denoting by $\mathcal{L}$ the Laplace transform, the following relations hold :

$$\begin{aligned}
\mathcal{L}(y)(x) &= \int_0^{+\infty} e^{-x.u} y(u)du \\
\mathcal{L}^{-1}(g)(x) &= \frac{1}{2i\pi}\int_C e^{z.x} g(z)dz
\end{aligned}$$

where $C$ is a closed contour enclosing the poles of `g`.
Input :

```
laplace(sin(x))
```

The expression (here $\sin(x)$) is an expression of the current variable (here $x$) and the answer will also be an expression of the current variable $x$.
Output :

```
1/((-x)^2+1)
```

Or input :

```
laplace(sin(t),t)
```

here the variable name is $t$ and this name is also used in the answer.
Output :

```
1/((-t)^2+1)
```

Or input :

```
laplace(sin(t),t,s)
```

here the variable name is $t$ and the variable name of the answer is $s$.
Output:

```
1/((-s)^2+1)
```

The following properties hold :

$$\begin{aligned}
\mathcal{L}(y')(x) &= -y(0) + x.\mathcal{L}(y)(x) \\
\mathcal{L}(y'')(x) &= -y'(0) + x.\mathcal{L}(y')(x) \\
&= -y'(0) - x.y(0) + x^2.\mathcal{L}(y)(x)
\end{aligned}$$

If $y\prime\prime(x) + p.y\prime(x) + q.y(x) \ = \ f(x)$, then :

$$\begin{aligned}
\mathcal{L}(f)(x) &= \mathcal{L}(y'' + p.y' + q.y)(x) \\
&= -y'(0) - x.y(0) + x^2.\mathcal{L}(y)(x) - p.y(0) + p.x.\mathcal{L}(y)(x)) + q.\mathcal{L}(y)(x) \\
&= (x^2 + p.x + q).\mathcal{L}(y)(x) - y'(0) - (x+p).y(0)
\end{aligned}$$

Therefore, if $a = y(0)$ et $b = y'(0)$, we have

$$\mathcal{L}(f)(x) = (x^2 + p.x + q).\mathcal{L}(y)(x) - (x+p).a - b$$

and the solution of the differential equation is :

$$y(x) = \mathcal{L}^{-1}((\mathcal{L}(f)(x) + (x+p).a + b)/(x^2 + p.x + q))$$

Example :
Solve :

$$y\prime\prime - 6.y\prime + 9.y \;=\; x.e^{3.x}, \quad y(0) = c\_0, \quad y\prime(0) = c\_1$$

Here, $p = -6,\; q = 9$.
Input :

```
laplace(x*exp(3*x))
```

Output :

```
1/(x^ 2-6*x+9)
```

Input :

```
ilaplace((1/(x^2-6*x+9)+(x-6)*c_0+c_1)/(x^2-6*x+9))
```

Output :

```
(216*x^3-3888*x*c_0+1296*x*c_1+1296*c_0)*exp(3*x)/1296
```

After simplification and factorisation (`factor` command) the solution $y$ is :

```
(-18*c_0*x+6*c_0+x^3+6*x*c_1)*exp(3*x)/6
```

Note that this equation could be solved directly, input :

```
desolve(y''-6*y'+9*y=x*exp(3*x),y)
```

Output :

```
exp(3*x)*(-18*c_0*x+6*c_0+x^3+6*x*c_1)/6
```

## 2.54 Other functions

### 2.54.1 Replace small values by 0: `epsilon2zero`

`epsilon2zero` takes as argument an expression of `x`.
`epsilon2zero` returns the expression where the values of modulus less than `epsilon` are replaced by zero. The expression is not evaluated.
The `epsilon` value is defined in the `cas` configuration (by default `epsilon=1e-10`).
Input :

```
epsilon2zero(1e-13+x)
```

Output (with `epsilon=1e-10`) :

```
0+x
```

Input :

```
epsilon2zero((1e-13+x)*100000)
```

Output (with `epsilon=1e-10`) :

```
(0+x)*100000
```

Input :

```
epsilon2zero(0.001+x)
```

Output (with `epsilon=0.0001`) :

```
0.001+x
```

### 2.54.2 List of variables : `lname indets`

`lname` (or `indets`) takes as argument an expression.
`lname` (or `indets`) returns the list of the symbolic variable names used in this expression.
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]
```

### 2.54.3 List of variables and of expressions : `lvar`

`lvar` takes as argument an expression.
`lvar` returns a list of variable names and non-rationnal expressions such that it's argument is a rational fraction with respect to the variables and expressions of the list.
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 :

```
[x,y,sqrt(z),sin(x)]
```

### 2.54.4 List of variables of an algebraic expressions: `algvar`

`algvar` takes as argument an expression.
`algvar` returns the list of the symbolic variable names used in this expression. The list is ordered by the algebraic extensions required to build the original expression.
Input :

```
algvar(y+x*sqrt(z))
```

Output :

```
[[y,x],[z]]
```

Input :

```
algvar(y*sqrt(x)*sqrt(z))
```

Output :

```
[[y],[z],[x]]
```

Input :

```
algvar(y*sqrt(x*z))
```

Output :

```
[[y],[x,z]]
```

Input :

```
algvar(y+x*sqrt(z)+y*sin(x))
```

Output :

```
[[x,y,sin(x)],[z]]
```

### 2.54.5 Test if a variable is in an expression : `has`

`has` takes as argument an expression and the name of a variable.
`has` returns `1` if this variable is in this expression, and else returns `0`.
Input :

```
has(x*y*sin(x),y)
```

Output :

```
1
```

Input :

```
has(x*y*sin(x),z)
```

Output :

```
0
```

### 2.54.6 Numeric evaluation : `evalf`

`evalf` takes as argument an expression or a matrix.
`evalf` returns the numeric value of this expression or of this matrix.
Input :

```
evalf(sqrt(2))
```

Output :

```
1.41421356237
```

Input :

```
evalf([[1,sqrt(2)],[0,1]])
```

Output :

```
[[1.0,1.41421356237],[0.0,1.0]]
```

### 2.54.7 Rational approximation : `float2rational exact`

`float2rational` (or `exact`) takes as argument an expression.
`float2rational` returns a rational approximation of all the floating point numbers $r$ contained in this expression, such that $|r - \texttt{float2rational}(r)| < \epsilon$, where $\epsilon$ is defined by `epsilon` in the `cas` configuration (`Cfg` menu, or `cas_setup` command).
Input :

```
float2rational(1.5)
```

Output :

```
3/2
```

Input :

```
float2rational(1.414)
```

Output :

```
707/500
```

Input :

```
float2rational(0.156381102937*2)
```

Output :

```
5144/16447
```

Input :

```
float2rational(1.41421356237)
```

Output :

```
114243/80782
```

Input :

```
float2rational(1.41421356237^2)
```

Output :

```
2
```

# Chapter 3

# Graphs

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.

## 3.1 Graph and geometric objects attributes

There are two kinds of attributes: global attributes of a graphic scene and individual attributes.

### 3.1.1 Individual attributes

Graphic attributes are optional arguments of the form `display=value`, they must be given as 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 added, e.g.
`plotfunc(`$x^2+y^2$`,[x,y],display=red+line_width_3+filled`

- Colors `display=` or `color=`
  - `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 garanteed to be portable.
- Point shapes `display=` one of the following value `rhombus_point plus_point square_point cross_point triangle_point star_point point_point invisible_point`
- Point width: `display=` one of the following value `point_width_n` where `n` is an integer between 1 and 7

- Line thickness: `thickness=n` or `display=line_width_n` where `n` is an integer between 1 and 7 or
- Line shape: `display=` one of the following value `dash_line solid_line dashdot_line dashdotdot_line cap_flat_line cap_square_line cap_round_line`
- Legend, value: `legend="legendname"`; position: `display=` one of `quandrant1 quadrant2 quadrant3 quadrant4` corresponding to the position of the legend of the object (using the trigonometric plan 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** Input

```
polygon(-1,-i,1,2*i,legend="P")
```

Input

```
point(1+i,legend="hello")
```

Input

```
A:=point(1+i);B:=point(-1);display(D:=droite(A,B),hidden_name)
```

Input

```
color(segment(0,1+i),red)
```

Input

```
segment(0,1+i,color=red)
```

### 3.1.2 Global attributes

These attributes are shared by all objets of the same scene

- `title="titlename"` defines the title
- `labels=["xname","yname","zname"]`: names of the $x, y, z$ axis
- `gl_x_axis_name="xname",gl_y_axis_name="yname",gl_z_axis_name=""`: individual definition of the names of the $x, y, z$ axis
- `legend=["xunit","yunit","zunit"]`: units for the $x, y, z$ axis
- `gl_x_axis_unit="xunit",gl_y_axis_unit="yunit",gl_z_axis_unit=""`: individual definition of the units of the $x, y, z$ axis

- `axes=true or false` show or hide axis
- `gl_texture="filename"`: background image
- `gl_x=xmin..xmax`, `gl_y=ymin..ymax`, `gl_z=zmin..zmax`: set the graphic configuration (do not use for interactive scenes)
- `gl_xtick=`, `gl_ytick=`, `gl_ztick=`: set the tick mark for the axis
- `gl_shownames=true or false`: show or hide objects names
- `gl_rotation=[x,y,z]`: defines the rotation axis for the animation rotation of 3-d scenesé
- `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

```
legend=["mn","kg"]
```

Input

```
titre="median_line";triangle(-1-i,1,1+i);median_line(-1-i,1,1+i);median_line(
```

Input

```
labels=["u","v"];plotfunc(u+1,u)
```

## 3.2 Graph of a function : `plotfunc funcplot DrawFunc Graph`

### 3.2.1 2-d graph

`plotfunc(f(x),x)` draws the graph of $y = f(x)$ for $x$ in the default interval, `plotfunc(f(x),x=a..b)` draws the graph of $y = f(x)$ for $a \leq x \leq b$. `plotfunc` accepts an optional `xstep=...` argument to specify the discretisation step in $x$.
Input :

```
plotfunc(x^2-2)
```

or

```
plotfunc(a^2-2,a=-1..2)
```

Output :

```
the graph of y=x^2-2
```

Input :

```
plotfunc(x^2-2,x,xstep=1)
```

Output :

```
a polygonal line which is a bad representation of
                    y=x^2-2
```

It is also possible to specify the number of points used for the representation of the function with `nstep=` instead of `xstep=`. For example, input :

```
plotfunc(x^2-2,x=-2..3,nstep=30)
```

### 3.2.2 3-d graph

`plotfunc` takes two main arguments : an expression of two variables or a list of several expressions of two variables and the list of these two variables, where each variable may be replaced by an equality variable=interval to specify the range for this variable (if not specified, default values are taken from the graph configuration). `plotfunc` accepts two optional arguments to specify the discretisation step in $x$ and in $y$ by `xstep=...` and `ystep=...`. Alternatively one can specify the number of points used for the representation of the function with `nstep=` (instead of `xstep` and `ystep`).
`plotfunc` draws the surface(s) defined by $z =$ the first argument.
Input :

```
plotfunc( x^2+y^2,[x,y])
```

Output :

```
A 3D graph of z=x^2+y^2
```

Input :

```
plotfunc(x*y,[x,y])
```

Output :

```
The surface z=x*y, default ranges
```

Input :

```
plotfunc([x*y-10,x*y,x*y+10],[x,y])
```

Output :

```
The surfaces z=x*y-10, z=x*y and z=x*y+10
```

Input :

```
plotfunc(x*sin(y),[x=0..2,y=-pi..pi])
```

Output :

The surface $z = x*y$ for the specified ranges

Now an example where we specify the $x$ and $y$ discretisation step with `xstep` and `ystep`. Input :

```
plotfunc(x*sin(y),[x=0..2,y=-pi..pi],xstep=1,ystep=0.5)
```

Output :

A portion of surface $z = x * y$

Alternatively we may 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 :

A portion of surface $z = x * y$

**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)`

### 3.2.3 3-d graph with rainbow colors

`plotfunc` represents a pure imaginary expression `i*E` of two variables with a rainbow color depending on the value of `z=E`. This gives an easy way to find points having the same third coordinate.
The first arguments of `plotfunc` must be `i*E` instead of `E`, the remaining arguments are the same as for a real 3-d graph (cf 3.2.2) Input :

```
plotfunc(i*x*sin(y),[x=0..2,y=-pi..pi])
```

Output :

A piece of the surface $z = x * \sin(y)$ with rainbow colors

**Remark**
If you want the impression or the traduction Latex, you have to use :
`Menu▶print▶Print(with Latex)`.

### 3.2.4 4-d graph.

`plotfunc` represents a complex expression `E` (such that `re(E)` is not identically 0 on the discretisation mesh) by the surface `z=abs(E)` where `arg(E)` defines the color from the rainbow. This gives an easy way to see the points having the same argument. Note that if `re(E)==0` on the discretisation mesh, it is the surface `z=E/i` that is represented with rainbow colors (cf 3.2.3).
The first argument of `plotfunc` is `E`, the remaining arguments are the same as for a real 3-d graph (cf 3.2.2). Input :

```
plotfunc((x+i*y)^2,[x,y])
```

Output :

```
A graph 3D of z=abs((x+i*y)^2 with the same color for
          points having the same argument
```

Input :

```
plotfunc((x+i*y)^2x,[x,y], display=filled)
```

Output :

```
The same surface but filled
```

We may specify the range of variation of $x$ and $y$ and the number of discretisation points, input :

```
plotfunc((x+i*y)^2,[x=-1..1,y=-2..2],
      nstep=900,display=filled)
```

Output :

```
The specified part of the surface with x between -1
   and 1, y between -2 and 2 and with 900 points
```

## 3.3 2d graph for Maple compatibility : `plot`

`plot(f(x),x)` draws the graph of $y = f(x)$. The second argument may specify the range of values `x=xmin..xmax`. One can also plot a function instead of an expression using the syntax `plot(f,xmin..xmax)`. `plot` accepts an optional argument to specify the step used in $x$ for the discretisation with `xstep=` or the number of points of the discretization with `nstep=`.
Input :

```
plot(x^2-2,x)
```

Output :

```
the graph of y=x^2-2
```

Input :

```
plot(x^2-2,xstep=1)
```

or

```
plot(x^2-2,x,xstep=1)
```

Output :

```
a polygonal line which is a bad representation of
                    y=x^2-2
```

Input!

```
plot(x^2-2,x=-2..3,nstep=30)
```

## 3.4 3d surfaces for Maple compatibility `plot3d`

`plot3d` takes three arguments : a function of two variables or an expression of two variables or a list of three functions of two variables or a list of three expressions of two variables and the names of these two variables with an optional range (for expressions) or the ranges (for functions).
`plot3d(f(x,y),x,y)` (resp `plot3d([f(u,v),g(u,v),h(u,v)],u,v)`) draws the surface $z = f(x,y)$ (resp $x = f(u,v), y = g(u,v), z = h(u,v)$). The syntax `plot3d(f(x,y),x=x0..x1,y=y0..y1)` or `plot3d(f,x0..x1,y0..y1)` specifies which part of surface will be computed (otherwise default values are taken from the graph configuration).
Input :

```
plot3d(x*y,x,y)
```

Output :

The surface $z = x * y$

Input :

```
plot3d([v*cos(u),v*sin(u),v],u,v)
```

Output :

The cone $x = v * \cos(u), y = v * \sin(u), z = v$

Input :

```
plot3d([v*cos(u),v*sin(u),v],u=0..pi,v=0..3)
```

Output :

A portion of the cone $x = v * \cos(u), y = v * \sin(u), z = v$

## 3.5 Graph of a line and tangent to a graph

### 3.5.1 Draw a line : `line`

**See also : ??** and **??** for line usage in geometry and see **??** and **??** for axis.
`line` takes as argument cartesian(s) equation(s) :

- in 2D: one line equation,
- in 3D: two plane equations.

`line` defines and draws the corresponding line.
Input :

```
line(2*y+x-1=0)
```

Output :

```
the line 2*y+x-1=0
```

Input :

```
line(y=1)
```

Output :

```
the horizontal line y=1
```

Input :

```
line(x=1)
```

Output :

```
the vertical line x=1
```

Input :

```
line(x+2*y+z-1=0,z=2)
```

Output :

```
the line x+2*y+1=0 in the plane z=2
```

Input :

```
line(y=1,x=1)
```

Output :

```
the vertical line crossing through (1,1,0)
```

**Remark**
`line` defines an oriented line :

- when the 2D line is given by an equation, it is rewritten as "left_member-right_member=`ax+by+c=0`", this determinates its normal vector `[a,b]` and the orientation is given by the vector `[b,-a]`) (or its orientation is defined by the 3D cross product of its normal vectors (with third coordinate 0) and the vector [0,0,1]).
  For example `line(y=2*x)` defines the line `-2x+y=0` with as direction the vector `[1,2]` (or `cross([-2,1,0],[0,0,1])=[1,2,0]`).

- when the 3D line is given by two plane equations, it's direction is defined by the cross product of the normals to the planes (where the plane equation is rewritten as "left_member-right_member=`ax+by+cz+d=0`", so that the normal is `[a,b,c]`).
  For example the `line(x=y,y=z)` is the line `x-y=0,y-z=0` and its direction is :
  `cross([1,-1,0],[0,1,-1])=[1,1,1]`.

### 3.5.2 Draw an 2D horizontal line : `LineHorz`

`LineHorz` takes as argument an expression $a$.
`LineHorz` draws the horizontal line $y = a$.
Input :

```
LineHorz(1)
```

Output :

```
the line y=1
```

### 3.5.3 Draw a 2D vertical line : `LineVert`

`LineVert` takes as argument an expression $a$.
`LineVert` draws the vertical line $x = a$.
Input :

```
LineVert(1)
```

Output :

```
the line x=1
```

### 3.5.4 Tangent to a 2D graph : `LineTan`

`LineTan` takes two arguments : an expression $E_x$ of the variable $x$ and a value $x0$ of $x$.
`LineTan` draws the tangent at $x = x0$ to the graph of $y = E_x$.
Input :

```
LineTan(ln(x),1)
```

Output :

```
the line y=x-1
```

Input :

```
equation(LineTan(ln(x),1))
```

Output :

```
y=(x-1)
```

### 3.5.5 Tangent to a 2D graph : `tangent`

**See also : ??** for plane geometry and **??** for 3D geometry.
`tangent` takes two arguments : an geometric object and a point `A`.
`tangent` draws tangent(s) to this geometric object crossing through `A`. If the geometric object is the graph `G` of a 2D function, the second argument is either, a real number `x0`, or a point `A` on `G`. In that case `tangent` draws a tangent to this graph `G` crossing through the point `A` or through the point of abscissa `x0`.
For example, define the function `g`

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

then the graph `G={(x,y)`$\in \mathbb{R}^2$`, y=g(x)}` of $g$ and a point $A$ on the graph $G$:

```
G:=plotfunc(g(x),x);
A:=point(1.2,g(1.2));
```

If we want to draw the tangent at the point `A` to the graph `G`, we will input:

```
T:=tangent(G, A)
```

or :

```
T:=tangent(G, 1.2)
```

For the equation of the tangent line, input :

```
equation(T)
```

### 3.5.6 Intersection of a 2D graph with the axis

- The ordinate of the intersection of the graph of $f$ with the $y$-axis is returned by :

  ```
  f(0)
  ```

  indeed the point of 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 to solve the equation $f(x) = 0$.
  If the equation is polynomial-like, `solve` will find the exact values of the abscissa of these points. Input:

  ```
  solve(f(x),x)
  ```

  Otherwise, we can find numeric approximations of these abscissa. First look at the graph for an initial guess or a range with an intersection and refine with `fsolve`.

## 3.6 Graph of inequations with 2 variables : `plotinequation inequationplot`

`plotinequation([f1(x,y)<a1,...fk(x,y)<ak],[x=x1..x2,y=y1..y2])`
draws the points of the plane whose coordinates satisfy the inequations of 2 variables :

$$\left\{\begin{array}{lcl} f1(x,y) & < & a1 \\ & ... & \\ fk(x,y) & < & ak \end{array}\right. , \quad x1 \le x \le x2, y1 \le y \le y2$$

Input :

```
plotinequation(x^2-y^2<3,
[x=-2..2,y=-2..2],xstep=0.1,ystep=0.1)
```

Output :

```
the filled portion enclosing the origin and limited by
the hyperbola x^2-y^2=3
```

Input :

```
plotinequation([x+y>3,x^2<y],
[x-2..2,y=-1..10],xstep=0.2,ystep=0.2)
```

Output :

```
the filled portion of the plane defined by
-2<x<2,y<10,x+y>3,y>x^2
```

Note that if the ranges for $x$ and $y$ are not specified, `Xcas` takes the default values of `X-,X+,Y-,Y+` defined in the general graphic configuration (`Cfg`▶`Graphic configuration`).

## 3.7 Graph of the area below a curve : `plotarea areaplot`

- With two arguments, `plotarea` shades the area below a curve.
  `plotarea(f(x),x=a..b)` draws the area below the curve $y = f(x)$ for $a < x < b$, i.e. the portion of the plane defined by the inequations $a < x < b$ and $0 < y < f(x)$ or $0 > y > f(x)$ according to the sign of $f(x)$ .
  Input :

  ```
  plotarea(sin(x),x=0..2*pi)
  ```

  Output :

  ```
  the portion of plane locates in the two archs of
  sin(x)
  ```

- With four arguments, `plotarea` represents a numeric approximation of the area below a curve, according to a quadrature method from the following list: `trapezoid,rectangle_left,rectangle_right,middle_point`.
  For example `plotarea(f(x),x=a..b,n,trapezoid)` draws the area of $n$ trapezes : the third argument is an integer $n$, and the fourth argument is the name of the numeric method of integration when $[a, b]$ is cut into $n$ equal parts.
  Input :

  ```
  plotarea((x^2,x=0..1,5,trapezoid)
  ```

  If you want to display the graph of the curve in contrast (e.g. in bold red), input :

```
        plotarea(x^2,x=0..1,5,trapezoid);
    plot(x^2,x=0..1,display=red+line_width_3)
```

Output :

```
the 5 trapezes used in the trapezoid method to
             approach the integral
```

Input :

```
plotarea((x^2,x=0..1,5,middle_point)
```

Or with the graph of the curve in bold red, input :

```
      plotarea(x^2,x=0..1,5,middle_point);
    plot(x^2,x=0..1,display=red+line_width_3)
```

Output :

```
the 5 rectangles used in the middle_point method
            to approach the integral
```

## 3.8 Contour lines: `plotcontour` `contourplot` `DrwCtour`

`plotcontour(f(x,y),[x,y])` (or `DrwCtour(f(x,y),[x,y])` or `contourplot(f(x,y),[x,y])`) draws the contour lines of the surface defined by $z = f(x,y)$ for $z = -10$, $z = -8$, .., $z = 0$, $z = 2$, .., $z = 10$. You may specify the desired contour lines by a list of values of $z$ given as third argument.
Input :

```
plotcontour(x^2+y^2,[x=-3..3,y=-3..3],[1,2,3],
    display=[green,red,black]+[filled$3])
```

Output :

```
the graph of the three ellipses x^2-y^2=n for n=1,2,3;
 the zones between these ellipses are filled with the
              color green,red or black
```

Input :

```
plotcontour(x^2-y^2,[x,y])
```

Output :

```
the graph of 11 hyperbolas x^2-y^2=n for n=-10,-8,..10
```

If you want to draw the surface in 3-d representation, input `plotfunc(f(x,y),[x,y])`, see 3.2.2):

```
plotfunc( x^2-y^2,[x,y])
```

Output :

```
A 3D representation of z=x^2+y^2
```

## 3.9 2-d graph of a 2-d function with colors : `plotdensity densityplot`

`plotdensity(f(x,y),[x,y])` or `densityplot(f(x,y),[x,y])` draws the graph of $z = f(x, y)$ in the plane where the values of $z$ are represented by the rainbow colors. The optional argument `z=zmin..zmax` specifies the range of $z$ corresponding to the full rainbow, if it is not specified, it is deduced from the minimum and maximum value of $f$ on the discretisation. The discretisation may be specified by optional `xstep=...` and `ystep=...` or `nstep=...` arguments.
Input :

```
plotdensity(x^2-y^2,[x=-2..2,y=-2..2],
        xstep=0.1,ystep=0.1)
```

Output :

```
A 2D graph where each hyperbola defined by x^2-y^2=z
          has a color from the rainbow
```

**Remark** : A rectangle representing the scale of colors is displayed below the graph.

## 3.10 Implicit graph: `plotimplicit implicitplot`

`plotimplicit` or `implicitplot` draws curves or surfaces defined by an implicit expression or equation. If the option `unfactored` is given as last argument, the original expression is taken unmodified. Otherwise, the expression is normalized, then replaced by the factorization of the numerator of it's 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 that 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.

### 3.10.1 2D implicit curve

- `plotimplicit(f(x,y),x,y)` draws the graphic representation of the curve defined by the implicit equation $f(x, y) = 0$ when $x$ (resp $y$) is in `WX-, WX+` (resp in `WY-, WY+`) defined by `cfg`,
- `plotimplicit(f(x,y),x=0..1,y=-1..1)` draws the graphic representation of the curve defined by the implicit equation $f(x, y) = 0$ when $0 \leq x \leq 1$ and $-1 \leq y \leq 1$

It is possible to add two arguments to specify the discretisation steps for $x$ and $y$ with `xstep=...` and `ystep=...`.
Input :

```
plotimplicit(x^2+y^2-1,x,y)
```

Or :

```
plotimplicit(x^2+y^2-1,x,y,unfactored)
```

Output :

```
The unit circle
```

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 :

```
The unit circle
```

Input :

```
plotimplicit(x^2+y^2-1,x=-2..2,y=-2..2,
            xstep=0.2,ystep=0.3)
```

Output :

```
The unit circle
```

### 3.10.2 3D implicit surface

- `plotimplicit(f(x,y,z),x,y,z)` draws the graphic representation of the surface defined by the implicit equation $f(x, y, z) = 0$,
- `plotimplicit(f(x,y,z),x=0..1,y=-1..1,z=-1..1)` draws the surface defined by the implicit equation $f(x, y, z) = 0$, where $0 \leq x \leq 1$, $-1 \leq y \leq 1$ and $-1 \leq z \leq 1$.

It is possible to add three arguments to specify the discretisation steps used for $x$, $y$ and $z$ with `xstep=...`, `ystep=...` and `zstep=...`.
Input :

```
  plotimplicit(x^2+y^2+z^2-1,x,y,z,
    xstep=0.2,ystep=0.1,zstep=0.3)
```

Input :

```
      plotimplicit(x^2+y^2+z^2-1,x,y,z,
xstep=0.2,ystep=0.1,zstep=0.3,unfactored)
```

Output :

```
The unit sphere
```

Input :

```
plotimplicit(x^2+y^2+z^2-1,x=-1..1,y=-1..1,z=-1..1)
```

Output :

```
The unit sphere
```

## 3.11 Parametric curves and surfaces : `plotparam paramplot DrawParm`

### 3.11.1 2D parametric curve

`plotparam([f(t),g(t)],t)` or `plotparam(f(t)+i*g(t),t)` (resp `plotparam(f(t)+i*g(t),t=t1..t2))` draws the parametric representation of the curve defined by $x = f(t), y = g(t)$ with the default range of values of $t$ (resp for $t1 \leq t \leq t2$).
The default range of values is taken as specified in the graphic configuration (`t-` and `t+`, cf. **??**). `plotparam` accepts an optional argument to specify the discretisation step for $t$ with `tstep=`.
Input :

```
plotparam(cos(x)+i*sin(x),x)
```

or :

```
plotparam([cos(x),sin(x)],x)
```

Output :

```
The unit circle
```

If in the graphic configuration `t` goes from -4 to 1, input :

```
plotparam(sin(t)+i*cos(t))
```

or :

```
plotparam(sin(t)+i*cos(t),t=-4..1)
```

or :

```
plotparam(sin(x)+i*cos(x),x=-4..1)
```

Output :

```
the arc (sin(-4)+i*cos(-4),sin(1)+i*cos(1)) of the
                    unit circle
```

If in the graphic configuration `t` goes from -4 to 1, input :

```
plotparam(sin(t)+i*cos(t),t,tstep=0.5)
```

Or :

```
plotparam(sin(t)+i*cos(t),t=-4..1,tstep=0.5)
```

Output :

```
           A polygon approching the arc
(sin(-4)+i*cos(-4),sin(1)+i*cos(1)) of the unit circle
```

### 3.11.2 3D parametric surface : `plotparam paramplot DrawParm`

`plotparam` takes two main arguments, a list of three expressions of two variables and the list of these variable names where each variable name may be replaced by variable=interval to specify the range of the parameters. It accepts an optionnal argument to specify the discretisation steps of the parameters $u$ and $v$ with `ustep=...` and `vstep=...`.
`plotparam([f(u,v),g(u,v),h(u,v)],[u,v])` draws the surface defined by the first argument : $x = f(u,v), y = g(u,v), z = h(u,v)$, where $u$ and $v$ ranges default to the graphic configuration.
Input :

```
plotparam([v*cos(u),v*sin(u),v],[u,v])
```

Output :

`The cone` $x = v * \cos(u), y = v * \sin(u), z = v$

To specify the range of each parameters, replace each variable by an equation variable=range, like this:

```
plotparam([v*cos(u),v*sin(u),v],[u=0..pi,v=0..3])
```

Output :

`A portion of the cone` $x = v * \cos(u), y = v * \sin(u), z = v$

Input :

```
plotparam([v*cos(u),v*sin(u),v],[u=0..pi,v=0..3],ustep=0.5,vstep=0.5)
```

Output :

`A portion of the cone` $x = v * \cos(u), y = v * \sin(u), z = v$

## 3.12 Curve defined in polar coordinates : `plotpolar polarplot DrawPol courbe_polaire`

Let $E_t$ be an expression depending of the variable $t$.
`plotpolar(`$E_t$`,t)` draws the polar representation of the curve defined by $\rho = E_t$ for $\theta = t$, that is in cartesian coordinates the curve $(E_t \cos(t), E_t \sin(t))$. The range of the parameter may be specified by replacing the second argument by `t=tmin..tmax`. The discretisation parameter may be specified by an optional `tstep=...` argument.
Input

```
plotpolar(t,t)
```

Output :

`The spiral` $\rho$`=t is plotted`

Input

```
plotpolar(t,t,tstep=1)
```

or :

```
plotpolar(t,t=0..10,tstep=1)
```

Output :

```
A polygon line approaching the spiral ρ=t is plotted
```

## 3.13 Graph of a recurrent sequence : `plotseq seqplot graphe_suite`

Let $f(x)$ be an expression depending of the variable $x$ (resp. $f(t)$ an expression depending of the variable $t$).
`plotseq(`$f(x)$`,a,n)` (resp. `plotseq(`$f(t)$`,t=a,n))` draws the line $y = x$, the graph of $y = f(x)$ (resp $y = f(t)$) and the $n$ first terms of the recurrent sequence defined by : $u_0 = a,\ \ u_n = f(u_{n-1})$. The $a$ value may be replaced by a list of 3 elements, $[a, x_-, x_+]$ where $x_-..x_+$ will be passed as $x$ range for the graph computation.
Input :

```
plotseq(sqrt(1+x),x=[3,0,5],5)
```

Output :

```
the graph of y=sqrt(1+x), of y=x and of the 5 first
terms of the sequence u_0=3 and u_n=sqrt(1+u_(n-1))
```

## 3.14 Tangent field : `plotfield fieldplot`

- Let $f(t, y)$ be an expression depending of two variables $t$ and $y$, then :

  ```
  plotfield(f(t,y),[t,y])
  ```

  draws the tangent field of the differential equation $y' = f(t, y)$ where $y$ is a real variable and where $t$ is the abscissa,

- Let $V$ be a vector of two expressions depending of 2 variables $x, y$ but independant of the time $t$, then

  ```
  plotfield(V,[x,y])
  ```

  draws the vector field $V$,

- The range of values of $t, y$ or of $x, y$ can be specified with
  `t=tmin..tmax`, `x=xmin..xmax`, `y=ymin..ymax`
  in place of the variable name.

- The discretisation may be specified with optional arguments `xstep=...`, `ystep=...`.

Input :

```
plotfield(4*sin(t*y),[t=0..2,y=-3..7])
```

Output :

```
Segments with slope 4*sin(t*y), representing tangents,
          are plotting in different points
```

With two variables $x, y$, input :

```
plotfield(5*[-y,x],[x=-1..1,y=-1..1])
```

## 3.15 Plotting a solution of a differential equation : `plotode` `odeplot`

Let $f(t, y)$ be an expression depending of two variables $t$ and $y$.

- `plotode(`$f(t,y)$`,[t,y],[t0,y0])` draws the solution of the differential equation $y' = f(t, y)$ crossing through the point `(t0,y0)` (i.e. such that $y(t_0) = y_0$)
- By default, $t$ goes in both directions. The range of value of $t$ may be specified by the optional argument `t=tmin..tmax`.
- We can also represent, in the space or in the plane, the solution of a differential equation $y' = f(t, y)$ where $y = (X, Y)$ is a vector of size 2. Just replace $y$ by the variable names $X, Y$ and the initial value $y_0$ by the two initial values of the variables at time $t_0$.

Input :

```
plotode(sin(t*y),[t,y],[0,1])
```

Output :

The graph of the solution of y'=sin(t,y) crossing through the point (0,1)

Input :

```
S:=odeplot([h-0.3*h*p, 0.3*h*p-p],
        [t,h,p],[0,0.3,0.7])
```

Output, the graph in the space of the solution of :

$$[h, p]' = [h - 0.3h * p, 0.3h * p - p] \quad [h, p](0) = [0.3, 0.7]$$

To have a 2-d graph (in the plane), use the option `plane`

```
S:=odeplot([h-0.3*h*p, 0.3*h*p-p],
      [t,h,p],[0,0.3,0.7],plane)
```

To compute the values of the solution, see the section **??**.

## 3.16 Interactive plotting of solutions of a differential equation : `interactive_plotode interactive_odeplot`

Let $f(t, y)$ be an expression depending of two varaiables $t$ and $y$.
`interactive_plotode(f(t,y),[t,y])` draws the tangent field of the differential equation $y' = f(t, y)$ in a new window. In this window, one can click on a point to get the plot of the solution of $y' = f(t, y)$ crossing through this point. You can further click to display several solutions. To stop press the `Esc` key.
Input :

```
interactive_plotode(sin(t*y),[t,y])
```

Output :

```
The tangent field is plotted with the solutions of
y'=sin(t,y) crossing through the points defined by
                   mouse clicks
```

## 3.17 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 ▶| (at the left of `Menu`).
- The display time of each graphic objet is specified in `animate` of the graph configuration (`cfg` button). Put a small time, to have a fast animation.
- If `animate` is `0`, the animation is frozen, you can move in the sequence of objects one by one by clicking on the mouse in the graphic scene.

### 3.17.1 Animation of a 2D graph : `animate`

`animate` can create a 2-d animation with graphs of functions depending of a parameter. The parameter is specified as the third argument of `animate`, the number of pictures as fourth argument with `frames`=number, the remaining arguments are the same as those of the `plot` command, see section 3.3, p. 332.
Input :

```
animate(sin(a*x),x=-pi..pi,a=-2..2,frames=10,color=red)
```

Output :

```
a sequence of graphic representations of y=sin(ax) for
           11 values of a between -2 and 2
```

### 3.17.2 Animation of a 3D graph : `animate3d`

`animate3d` can create a 3-d animation with function graphs depending of a parameter. The parameter is specified as the third argument of `animate3d`, the number of pictures as fourth argument with `frames`=number, the remaining arguments are the same as those of the `plotfunc` command, see section 3.2.2, p. 330.
Input :

```
animate3d(x^2+a*y^2,[x=-2..2,y=-2..2],a=-2..2,
          frames=10,display=red+filled)
```

Output :

```
a sequence of graphic representations of z=x^2+a*y^2
        for 11 values of a between -2 and 2
```

### 3.17.3 Animation of a sequence of graphic objects : `animation`

`animation` animates the representation of a sequence of graphic objects with a given display time. The sequence of objects depends most of the time of a parameter and is defined using the `seq` command but it is not mandatory.
`animation` takes as argument the sequence of graphic objects.
To define a sequence of graphic objects with `seq`, enter the definition of the graphic object (depending on the parameter), the parameter name, it's minimum value, it's maximum value maximum and optionnaly a step value.
Input :

```
animation(seq(plotfunc(cos(a*x),x),a,0,10))
```

Output :

The sequence of the curves defined by $y = \cos(ax)$, for
$a = 0, 1, 2..10$

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 sequence of the curves defined by $y = \cos(ax)$, for
$a = 0, 0.5, 1, 1.5..10$

Input :

```
animation(seq(plotfunc([cos(a*x),sin(a*x)],x=0..2*pi/a),
                          a,1,10))
```

Output :

The sequence of two curves defined by $y = \cos(ax)$ and $y = \sin(ax)$, for $a = 1..10$ and for $x = 0..2\pi/a$

Input :

```
animation(seq(plotparam([cos(a*t),sin(a*t)],
            t=0..2*pi),a,1,10))
```

Output :

The sequence of the parametric curves defined by $x = \cos(at)$ and $y = \sin(at)$, for $a = 1..10$ and for $t = 0..2\pi$

Input :

```
animation(seq(plotparam([sin(t),sin(a*t)],
      t,0,2*pi,tstep=0.01),a,1,10))
```

Output :

The sequence of the parametric curves defined by $x = \sin(t), y = \sin(at)$, for $a = 0..10$ and $t = 0..2\pi$

Input :

```
animation(seq(plotpolar(1-a*0.01*t^2,
      t,0,5*pi,tstep=0.01),a,1,10))
```

Output :

The sequence of the polar curves defined by $\rho = 1 - a * 0.01 * t^2$, for $a = 0..10$ and $t = 0..5\pi$

Input :

```
              plotfield(sin(x*y),[x,y]);
animation(seq(plotode(sin(x*y),[x,y],[0,a]),a,-4,4,0.5))
```

Output :

The tangent field of y'=sin(xy) and the sequence of the integral curves crossing through the point $(0, a)$ for $a$=-4,-3.5...3.5,4

Input :

```
animation(seq(display(square(0,1+i*a),filled),a,-5,5))
```

Output :

The sequence of the squares defined by the points 0 and 1+i*$a$ for $a = -5..5$

Input :

```
animation(seq(droite([0,0,0],[1,1,a]),a,-5,5))
```

Output :

```
The sequence of the lines defined by the points
        [0,0,0] and [1,1,a] for a = -5..5
```

Input :

```
animation(seq(plotfunc(x^2-y^a,[x,y]),a=1..3))
```

Output :

```
The sequence of the "3D" surface defined by x^2 - y^a, for
             a = 1..3 with rainbow colors
```

Input :

```
animation(seq(plotfunc((x+i*y)^a,[x,y],
           display=filled),a=1..10)
```

Output :

```
The sequence of the "4D" surfaces defined by (x + i * y)^a,
            for a = 0..10 with rainbow colors
```

**Remark** We may also define the sequence with a program, for example if we want to draw the segments of length 1, $\sqrt{2}$...$\sqrt{20}$ constructed with a right triangle of side 1 and the previous segment (note that there is a `c:=evalf(..)` statement to force approx. evaluation otherwise the computing time would be too long) :

```
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 input :

```
animation(seg(20))
```

We see, each point, one to one with a display time that depends of the `animate` value in `cfg`.
Or :

```
L:=seg(20); s:=segment(0,L[k])$(k=0..20)
```

We see 21 segments.
Then, input :

```
animation(s)
```

We see, each segment, one to one with a display time that depends of the `animate` value in `cfg`.

# Chapter 4

# Numerical computations

Real numbers may have an exact representation (e.g. rationals, symbolic expressions involving square roots or constants like $\pi$, ...) or approximate representation, which means that 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) optionnally 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.

## 4.1 Floating point representation.

In this section, we explain how real numbers are represented.

### 4.1.1 Digits

The `Digits` variable 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 created MPFR float is `ceil(Digits*log(10)/log(2))`.

Note that if you change the value of `Digits`, this will affect the creation of new real numbers compiled from commandlines 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.

### 4.1.2 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 $\alpha$ denotes the integer from 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 from 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**: $2^{-52} = 0.2220446049250313e - 15$

### 4.1.3 Examples of representations of normalized floats

- 3.1 :
  We have :

$$\begin{aligned} 3.1 &= 2 * (1 + \frac{1}{2} + \frac{1}{2^5} + \frac{1}{2^6} + \frac{1}{2^9} + \frac{1}{2^{10}} + ....) \\ &= 2 * (1 + \frac{1}{2} + \sum_{k=1}^{\infty} (\frac{1}{2^{4*k+1}} + \frac{1}{2^{4*k+2}})) \end{aligned}$$

  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)
```

### 4.1.4 Difference between the representation of (3.1-3) and of 0.1

- representation of 0.1 :
  We have :

$$0.1 = 2^{-4}*(1+\frac{1}{2}+\frac{1}{2^4}+\frac{1}{2^5}+\frac{1}{2^8}+\frac{1}{2^9}+...) = 2^{-4}*\sum_{k=0}^{\infty}(\frac{1}{2^{4*k}}+\frac{1}{2^{4*k+1}})$$

  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).

- representation of a:=3.1-3 :
  Computing a is done by adjusting exponents (here nothing to do), then substract the mantissa, and adjust the exponent of the result to have a normalized float. The exponent is $\alpha = -4$ (that corresponds at $2*2^{-5}$) and the bits corresponding to the mantissa begin at $1/2 = 2*2^{-6}$ : the bits of the mantissa are shifted to the left of 5 positions and we have :

```
3f (00111111), b9 (10111001), 99 (10011001), 99 (10011001),
99 (10011001), 99 (10011001), 99 (10011001), 9a (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

```
floor(1/(3.1-3))
```

returns `9` and not `10` when `Digits:=14`.

## 4.2 Approx. evaluation : `evalf approx` and `Digits`

`evalf` or `approx` evaluates to a numeric approximation (if possible).
Input :

```
evalf(sqrt(2))
```

Output, if in the `cas` configuration (`Cfg` menu) `Digits=7` (that is hardware floats are used, and 7 digits are displayed) :

```
1.414214
```

You can change the number of digits in a commandline 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
```

## 4.3 Numerical algorithms

### 4.3.1 Approximate solution of an equation : `newton`

`newton` takes as arguments : an expression `ex`, the variable name of this expression (by default `x`), and three values `a` (by default `a=0`), `eps` (by default `eps=1e-8`) and `nbiter` (by default `nbiter=12`).
`newton(ex,x,a,eps,nbiter)` computes an approximate solution `x` of the equation `ex=0` using the Newton algorithm with starting point `x=a`. The maximum number of iterations is `nbiter` and the precision is `eps`.
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
```

### 4.3.2 Approximate computation of the derivative number : `nDeriv`

`nDeriv` takes as arguments : an expression `ex`, the variable name of this expression (by default `x`), and `h` (by default `h=0.001`).
`nDeriv(ex,x,h)` computes an approximated value of the derivative of the expression `ex` at the point `x` and returns :

```
(f(x+h)-f(x+h))/2*h
```

Input :

```
nDeriv(x^ 2,x)
```

Output :

```
((x+0.001)^2-(x+-0.001)^2)*500.0
```

Input :

```
subst(nDeriv(x^ 2,x),x=1)
```

Output :

```
2
```

Input :

```
nDeriv(exp(x^ 2),x,0.00001)
```

Output :

```
(exp((x+1e-05)^2)-exp((x+-1e-05)^2))*50000
```

Input :

```
subst(exp(nDeriv(x^ 2),x,0.00001),x=1)
```

Output :

```
5.43656365783
```

which is an approximate value of `2e=5.43656365692`.

### 4.3.3 Approximate computation of integrals : `romberg nInt`

`romberg` or `nInt` takes as arguments : an expression `ex`, the variable name of this expression (by default `x`), and two real values `a,b`.
`romberg(ex,x,a,b)` or `nInt(ex,x,a,b)` computes an approximated value of the integral $\int_a^b ex\ dx$ using the Romberg method. The integrand must be sufficiently regular for the approximation to be accurate. Otherwise, `romberg` returns a list of real values, that comes 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-Mac Laurin formula to remove successive even powers of the step of the trapezoid rule).
Input :

```
romberg(exp(x^2),x,0,1)
```

Output :

```
1.46265174591
```

### 4.3.4 Approximate solution of y'=f(t,y) : `odesolve`

- Let $f$ be a function from $\mathbb{R}^2$ to $\mathbb{R}$.
  `odesolve(f(t,y),[t,y],[t0,y0],t1)` or
  `odesolve(f(t,y),t=t0..t1,y,y0)` or
  `odesolve(t0..t1,f,y0)` or
  `odesolve(t0..t1,(t,y)->f(t,y),y0)`
  returns an approximate value of $y(t1)$ where $y(t)$ is the solution of:
  $$y'(t) = f(t, y(t)), \quad y(t0) = y0$$
- `odesolve` accepts an optional argument for the discretisation of `t` (`tstep=value`). This value is passed as 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*(t1-t0)/tstep` (if the number of iterations exceeds this value, the solver will stopsat a time $t < t1$).

- `odesolve` accepts `curve` as an optional argument. In that case, `odesolve` returns the list of all the $[t, [y(t)]]$ values that where computed.

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 define the function :

```
f(t,y):=sin(t*y)
```

and input :

```
odesolve(0..2,f,1)
```

Output :

```
[1.82241255675]
```

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 :

```
[1.82241255675]
```

Input :

```
odesolve(sin(t*y),t=0..2,y,1,tstep=0.5,curve)
```

Output :

```
[[0.760963063136,[1.30972370515]],[1.39334557388,[1.86417104853]]]
```

### 4.3.5 Approximate solution of the system v'=f(t,v) : `odesolve`

- If $v$ is a vector of variables $[x1, .., xn]$ and if $f$ is given by a vector of expressions `[e1,...,en]` depending of $t$ and of $[x1, .., xn]$, if the init value of $v$ at `t0` is the vector $[x10, ..., xn0]$ then the instruction

  ```
  odesolve([e1,..,en],t=t0..t1,[x1,...,xn],
                    [x10,...,xn0])
  ```

  returns an approximated value of $v$ at $t = t1$. With the optional argument `curve`, `odesolve` returns the list of the intermediate values of $[t, v(t)]$ computed by the solver.

  Example, to 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 :

  ```
  [-1.79045146764e-15,-1]
  ```

- If $f$ is a function from $\mathbb{R} \times \mathbb{R}^n$ to $\mathbb{R}^n$.
  `odesolve(t0..t1,(t,v)->f(t,v),v0)` or
  `odesolve(t0..t1,f,v0)`
  computes an approximate value of $v(t1)$ where the vector $v(t)$ in $\mathbb{R}^n$ is the solution of

  $$v'(t) = f(t, v(t)), v(t0) = v0$$

  With the optional argument `curve`, `odesolve` returns the list of the intermediate value $[t, v(t)]$ computed by the solver.

  Example, to 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 define the function:

  ```
  f(t,v):=[-v[1],v[0]]
  ```

  then input :

```
odesolve(0..pi,f,[0,1])
```

Output :

```
[-1.79045146764e-15,-1]
```

Alternative input :

```
odesolve(0..pi/4,f,[0,1],curve)
```

Output :

```
[[0.1781,[-0.177159948386,0.984182072936]],
 [0.3781,[-0.369155338156,0.929367707805]],
  [0.5781,[-0.54643366953,0.837502384954]],
 [0.7781,[-0.701927414872,0.712248484906]]]
```

## 4.4 Solve equations with `fsolve nSolve`

`fsolve` or `nSolve` solves numeric equations (unlike `solve` or `proot`, it is not limited to polynomial equations) of the form:

$$f(x) = 0, \quad x \in ]a, b[$$

`fsolve` or de `nSolve` accepts a last optional argument, the name of an iterative algorithm to be used by the GSL solver. The different methods are explained in the following section.

### 4.4.1 `fsolve` or `nSolve` 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. We compute the middle point value. The function sign at this point, gives us the half-interval on which the next iteration will be performed.
Input :

```
fsolve((cos(x))=x,x,-1..  1,bisection_solver)
```

Output :

```
[0.739085078239,0.739085137844]
```

### 4.4.2 `fsolve` or `nSolve` 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.
Input :

```
fsolve((cos(x))=x,x,-1..1,brent_solver)
```

Output :

```
[0.73908513321 5,0.739085133215]
```

### 4.4.3 `fsolve` or `nSolve` with the option `falsepos_solver`

The "false position" algorithm is an iterative algorithm based on linear interpolation : we compute 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.
Input :

```
fsolve((cos(x))=x,x,-1..1,falsepos_solver)
```

Output :

```
[0.739085133215,0.739085133215]
```

### 4.4.4 `fsolve` or `nSolve` with the option `newton_solver`

`newton_solver` is the standard Newton method. The algorithm starts at an init value $x_0$, then we search 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.
Input :

```
fsolve((cos(x))=x,x,0,newton_solver)
```

Output :

```
0.739085133215
```

### 4.4.5 `fsolve` or `nSolve` with the option `secant_solver`

The secant methode is a simplified version of the Newton method. The computation of $x_1$ is done using the Newton method. The computation of $f'(x_n), n > 1$ is done approximatively. 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 $(1 + \sqrt{5})/2 \approx 1.62...$.
Input :

```
fsolve((cos(x))=x,x,-1..1,secant_solver)
```

Output :

```
[0.739085078239,0.739085137844]
```

Input :

```
fsolve((cos(x))=x,x,0,secant_solver)
```

Output :

```
0.739085133215
```

### 4.4.6 `fsolve` or `nSolve` with the option `steffenson_solver`

The Steffenson method is generically the fastest method.
It combines the Newton method with a "delta-two" Aitken acceleration : with the Newton method, we obtain the sequence $x_i$ and the convergence acceleration gives the Steffenson sequence

$$R_i = x_i - \frac{(x_{i+1} - x_i)^2}{(x_{i+2} - 2x_{i+1} + x_i)}$$

Input :

```
fsolve(cos(x)=x,x,0,steffenson_solver)
```

Output :

```
0.739085133215
```

## 4.5 Solve systems with `fsolve`

`Xcas` provides six methods (inherited from the GSL) to solve numeric systems of equations of the form $f(x) = 0$:

- Three methods use the jacobian matrix $f'(x)$ and their names are terminated with `j_solver`.
- The three other methods use approximation for $f'(x)$ and use only $f$.

All methods use an iteration of Newton kind

$$x_{n+1} = x_n - f'(x_n)^{-1} * f(x_n)$$

The four methods `hybrid*_solver` use also a method of gradient descent when the Newton iteration would make a too large 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`.

### 4.5.1 `fsolve` with the option `dnewton_solver`

Input :

```
fsolve([x^2+y-2,x+y^2-2],[x,y],[2,2],dnewton_solver)
```

Output :

```
[1.0,1.0]
```

### 4.5.2 `fsolve` with the option `hybrid_solver`

Input :

```
fsolve([x^2+y-2,x+y^2-2],[x,y],[2,2],
    cos(x)=x,x,0,hybrid_solver)
```

Output :

```
[1.0,1.0]
```

### 4.5.3 `fsolve` with the option `hybrids_solver`

Input :

```
fsolve([x^2+y-2,x+y^2-2],[x,y],[2,2],hybrids_solver)
```

Output :

```
[1.0,1.0]
```

### 4.5.4 `fsolve` with the option `newtonj_solver`

Input :

```
fsolve([x^2+y-2,x+y^2-2],[x,y],[0,0],newtonj_solver)
```

Output :

```
[1.0,1.0]
```

### 4.5.5 `fsolve` with the option `hybridj_solver`

Input :

```
fsolve([x^2+y-2,x+y^2-2],[x,y],[2,2],hybridj_solver)
```

Output :

```
[1.0,1.0]
```

### 4.5.6 `fsolve` with the option `hybridsj_solver`

Input :

```
fsolve([x^2+y-2,x+y^2-2],[x,y],[2,2],hybridsj_solver)
```

Output :

```
[1.0,1.0]
```

## 4.6 Numeric roots of a polynomial : `proot`

`proot` takes as argument a squarefree polynomial, either in symbolic form or as a list of polynomial coefficients (written by decreasing order).
`proot` returns a list of the numeric roots of this polynomial.
To find the numeric roots of $P(x) = x^3 + 1$, input :

```
proot([1,0,0,1])
```

or :

```
proot(x^3+1)
```

Output :

```
[0.5+0.866025403784*i,0.5-0.866025403784*i,-1.0]
```

To find the numeric roots of $x^2-3$, input :

```
proot([1,0,-3])
```

or :

```
proot(x^2-3)
```

Output :

```
[1.73205080757,-1.73205080757]
```

## 4.7 Numeric factorization of a matrix : `cholesky qr lu svd`

Matrix numeric factorizations of

- Cholesky,
- QR,
- LU,
- svd,

are described in section 2.48.

# Chapter 5

# Unit objects and physical constants

The `Phys` menu contains:

- the physical constants (`Constant` sub-menu),
- the unit conversion functions (`Unit_convert` sub-menu),
- the unit prefix (`Unit_prefix` sub-menu)
- the unit objects organized by subject

## 5.1 Unit objects

### 5.1.1 Notation of unit objects

A unit object has two parts : a real number and a unit expression (a single unit or multiplicative combinaison of unit). 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)`.
If a prefix is put before the unit then the unit is multiplicated by a power of 10. For example `k` or `K` for kilo (indicate a multiplication by $10^3$), `D` for deca (indicate a multiplication by 10), `d` for deci (indicate a multiplication by $10^{-1}$) etc...
Input :

```
10.5_m
```

Output :

```
a unit object of value 10.5 meters
```

Input :

```
10.5_km
```

Output :

```
a unit object of value 10.5 kilometers
```

### 5.1.2 Computing with units

`Xcas` performs usual arithmetic operations (+, -, *, /, ^) on unit objects. Different units may be used, but they must be compatible for + and -. 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)` or `_(u1/u2)`.
- for an addition or a substraction of compatible unit objects, the result is expressed with the same unit as the first term of the operation.

Input :

```
1_m+100_cm
```

Output :

```
2_m
```

Input :

```
100_cm+1_m
```

Output :

```
200_cm
```

Input :

```
1_m*100_cm
```

Output :

```
1_m^2
```

### 5.1.3 Convert units into MKSA units : `mksa`

`mksa` converts a unit object into a unit object written with the compatible `MKSA` base unit.
Input :

```
mksa(15_C)
```

Output :

```
15_(s*A)
```

### 5.1.4 Convert units : `convert`

`convert` convert units : the first argument is an unit object and the second argument is the new unit (which must be compatible).
Input :

```
convert(1_h,_s)
```

Output :

```
3600_s
```

Input :

```
convert(3600_s,_h)
```

Output :

```
1_h
```

### 5.1.5 Factorize an unit : `ufactor`

`ufactor` factorizes an unit in a unit object : the first argument is an unit object and the second argument is the unit to factorize.
The result is an unit object multiplicated by the remaining `MKSA` units.
Input :

```
ufactor(3_J,_W)
```

Output :

```
3_(W*s)
```

Input :

```
ufactor(3_W,_J)
```

Output :

```
3_(J/s)
```

### 5.1.6 Simplify an unit : `usimplify`

`usimplify` simplifies an unit in an unit object.
Input :

```
usimplify(3_(W*s))
```

Output :

```
3_J
```

### 5.1.7 Unit prefixes

You can insert a unit prefix in front of a unit to indicate a power of ten.
The following tables gives the available prefixes:

| Prefix | Name | (*10^) n | Prefix | Name | (*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 |

**Remark**
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`.

## 5.2 Constants

### 5.2.1 Notation of physical constants

If you want to use a physical constants inside Xcas, put it's name between two characters _ ("underscore"). Don't confuse physical constants with symbolic constants, for example, $e, \pi$ are symbolic constants as `_c_,` `_NA_` are physical constants.
Input :

```
_c_
```

Output speed of light in vacuum :

```
299792458_m*s^-1
```

Input :

```
_NA_
```

Output Avogadro's number :

```
6.0221367e23_gmol^-1
```

### 5.2.2 Constants Library

The physical constants are in the `Phys` menu, `Constant` sub-menu. The following table gives the Constants Library :

| Name | Description |
|---|---|
| `_NA_` | Avogadro's number |
| `_k_` | Boltzmann constant |
| `_Vm_` | Molar volume |
| `_R_` | Universal gas constant |
| `_StdT_` | Standard temperature |
| `_StdP_` | Standard pressure |
| `_sigma_` | Stefan-Boltzmann constant |
| `_c_` | Speed of light in vacuum |
| `_epsilon0_` | Permittivity of vacuum |
| `_mu0_` | Permeability of vacuum |
| `_g_` | Acceleration of gravity |
| `_G_` | Gravitational constant |
| `_h_` | Planck's constant |
| `_hbar_` | Dirac's constant |
| `_q_` | Electron charge |
| `_me_` | Electron rest mass |
| `_qme_` | q/me (Electron charge/masse) |
| `_mp_` | Proton rest mass |
| `_mpme_` | mp/me (proton masse/electron masse) |
| `_alpha_` | Fine structure constant |
| `_phi_` | Magnetic flux quantum |
| `_F_` | Faraday constant |
| `_Rinfinity_` | Rydberg constant |
| `_a0_` | Bohr radius |
| `_muB_` | Bohr magneton |
| `_muN_` | Nuclear magneton |
| `_lambda0_` | Photon wavelength (ch/e) |
| `_f0_` | Photon frequency (e/h) |
| `_lambdac_` | Compton wavelength |
| `_rad_` | 1 radian |
| `_twopi_` | 2*pi radians |
| `_angl_` | 180 degrees angle |
| `_c3_` | Wien displacement constant |
| `_kq_` | k/q (Boltzmann/electron charge) |
| `_epsilon0q_` | epsilon0/q (permitivity /electron charge) |
| `_qepsilon0_` | q*epsilon0 (electron charge *permitivity) |
| `_epsilonsi_` | Silicium dielectric constant |
| `_epsilonox_` | Bioxyd of silicium dielectric constant |
| `_I0_` | Reference intensity |

To have the value of a constant, input the constant name in the command line of `Xcas` and valid with `enter` (don't forget to put _ at the begining and at the end of the constant name).