Un logiciel de calcul formel permet de manipuler des objets mathématiques de manière exacte. On peut classer les logiciels de calcul formel en deux grandes catégories

— les commerciaux, dont les plus connus sont Maple et Mathematica, Ces logiciels ne sont plus disponibles à l'oral de modélisation de l'agrégation de maths depuis la session 2015.

— les libres : certains sont généralistes comme Xcas, Maxima, Sage, d'autres plus spécialisés comme Gap (groupes), Pari/Gp (théorie des nombres), etc... Leur utilisation est bien sur gratuite.

A partir de 2015, Xcas, Maxima et Sage sont les seuls logiciels de calcul formel utilisables à l'épreuve de modélisation.

Les sections 1 et 2 de ce document sont un TP de prise en main suivi par un TP sur les structures de données et algorithmes fondementaux. Les sections suivantes de ce document sont disponibles en ligne :
`www-fourier.ujf-grenoble.fr/~parisse/agreg/agregtp1.pdf`
il s'agit d'une version abrégée du tutoriel de Xcas.

# 1 TP de prise en main

Une fois identifié sur les PC de la salle TP, Xcas se lance depuis un terminal (menu Système ou Accessoires ou clic droit souris) par la commande `xcas &` ou depuis le menu Education du menu général. Vous pouvez aussi utiliser la version HTML5 de Xcas depuis Firefox ou un navigateur compatible (sur PC/Mac, tablette, voire même sur smartphone), allez sur
`http://www-fourier.ujf-grenoble.fr/~parisse/xcasfr.html`
Pour installer Xcas sur votre ordinateur personnel, allez sur le site
`http://www-fourier.ujf-grenoble.fr/~parisse/install_fr.html`
Ensuite pour lancer Xcas, cela dépend de votre système d'exploitation (linux debian : comme en salle TP, windows : icone Xcas du bureau, mac ouvrir Xcas depuis le Finder dans Applications, usr, bin puis conservez-le dans le dock).

Le jour de l'épreuve de modélisation, vous pourrez lancer Xcas version desktop depuis son icone. Pour lancer Xcas version Firefox, clonez une session vide (menu Fichier, Clone offline). **Vérifiez la configuration du logiciel dans la barre d'état, et modifiez-la si nécessaire.**

Les commandes les plus utiles de Xcas se trouvent dans le menu Outils, le menu Expressions permet de réécrire des expressions, le menu Cmd comporte une liste bien plus exhaustives de commandes, le menu Prog des assistants et commandes de programmation, le menu Graphe des commandes et assistants pour faire des représentations graphiques.

Le menu Aide, Index affiche une aide courte sur une commande et des exemples recopiables que l'on peut ensuite modifier, on peut aussi lancer l'aide en tapant la touche F1 lorsque le curseur se trouve après un nom de commande (ou un début de nom de commande), le bouton Details permet d'afficher une aide plus complète. Vous pouvez aussi rechercher un ou plusieurs mots clefs dans la documentation (menu Aide). Notez que le manuel Algorithmes (menu Aide, Manuels) contient un sur-ensemble du cours de l'option C et du programme commun, il est disponible en deux versions : PDF imprimable ou HTML5 avec possibilité de modifier et exécuter des exemples de calculs. Le manuel de programmation contient aussi des exemples de programmes en relation avec le programme.

Si vous n'avez jamais utilisé de logiciel de calcul formel, vous pouvez commencer par parcourir le tutoriel de Xcas (menu Aide, Débuter en calcul formel, Tutoriel) ou la version en ligne de ce tutoriel (sections 3 et suivantes) ou/et vous inspirer des exemples de la section guide de survie du manuel Algorithmes (menu Aide, Manuels, Algorithmes).

Pour saisir une matrice (liste de listes de même taille), on peut utiliser la commande `matrix`, ou entrer les coefficients un par un en ligne de commandes (par exemple `[[1,a],[a,1]]`) ou avec le tableur (Tableur, nouveau tableur, donner un nom de variable et entrer les coefficients).

Pour commencer à écrire des programmes, utiliser Nouveau programme dans le menu Prg, puis les assistants de création de fonction, test et boucle. Vous pouvez utiliser la syntaxe en français (Xcas) ou compatible Python, les assistants s'adaptent au mode choisi dans la configuration du CAS. La commande `debug` permet de mettre au point un programme en l'exécutant instruction par instruction et en visualisant la valeur des variables.

1. Écrire le polynôme $(x+3)^7 \times (x-5)^6$ selon les puissances décroissantes de $x$.
2. Simplifier les expressions suivantes :
$$\sqrt{3+2\sqrt{2}}, \quad \frac{1+\sqrt{2}}{1+2\sqrt{2}}, \quad e^{i\pi/6}, \quad 4\text{atan}(\frac{1}{5}) - \text{atan}(\frac{1}{239})$$
3. Factoriser :
$$x^8 - 3x^7 - 25x^6 + 99x^5 + 60x^4 - 756x^3 + 1328x^2 - 960x + 256$$
$$x^6 - 2x^3 + 1, \quad (-y+x)z^2 - xy^2 + x^2y$$
4. Calculez les intégrales et simplifiez le résultat :
$$\int \frac{1}{e^x - 1}\, dx, \quad \int \frac{1}{x\ln(x)} \ln(\ln(x))dx, \quad \int e^{x^2}\, dx, \quad \int x\sin(x)e^x\, dx$$
Vérifiez en dérivant les expressions obtenues.
5. Déterminer la valeur de :
$$\int_1^2 \frac{1}{(1+x^2)^3}, \quad \int_1^2 \frac{1}{x^3+1}\, dx$$
6. Calculer les sommes suivantes
$$\sum_{k=1}^{N} k, \ \sum_{k=1}^{N} k^2, \ \sum_{k=1}^{\infty} \frac{1}{k^2}$$
7. Développer $\sin(3x)$, linéariser l'expression obtenue et vérifier qu'on retrouve l'expression initiale.
8. Calculer le développement de Taylor en $x=0$ à l'ordre 4 de :
$$\ln(1+x+x^2), \quad \frac{\exp(\sin(x))-1}{x+x^2}, \quad \sqrt{1+e^x}, \quad \frac{\ln(1+x)}{\exp(x)-\sin(x)}$$
9. Trouver les entiers $n$ tels que le reste de la division euclidienne de $123n$ par 256 soit 17.
10. Déterminer la liste des diviseurs de 45768.
Factoriser 100 !
11. Résoudre le système linéaire :
$$\begin{cases} x & + & y & + & az & = & 1 \\ x & + & ay & + & z & = & 2 \\ ax & + & y & + & z & = & 3 \end{cases}$$
12. Déterminer l'inverse de la matrice :
$$A = \begin{pmatrix} 1 & 1 & 1 & a \\ 1 & 1 & a & 1 \\ 1 & a & 1 & 1 \\ a & 1 & 1 & 1 \end{pmatrix}$$

# 2 TP1 : structures de données, algorithmes fondamentaux

1. Algorithmes fondementaux : écrire des programmes implémentant
   (a) le pgcd de 2 entiers : avec le reste euclidien $a = bq + r, \quad r \in [0, b[$ puis avec la valeur absolue du reste symétrique $r \in ]-b/2, b/2]$. Comparer le nombre d'étapes pour les deux méthodes.
   (b) l'algorithme de Bézout
   (c) l'inverse modulaire en ne calculant que ce qui est nécessaire dans l'algorithme de Bézout
   (d) les restes chinois

   Vérifiez vos programmes en comparant avec les fonctions intégrées (`gcd, iegcd, inv(a % b), ichinrem`).
2. Faire une simulation illustrant le pourcentage asymptotique $\log_2((a+1)^2/((a+1)^2-1))$ (Knuth, The Art of Computer Programming, volume 2) de quotients égaux à $a$ pour $a = 1, 2, 3$ dans l'algorithme d'Euclide. On pourra utiliser `1+rand(N)` pour générer des entiers aléatoires compris entre 1 et $N$.
3. À quelle vitesse votre logiciel multiplie-t-il des grands entiers (en fonction du nombre de chiffres) ? On pourra tester le temps de calcul du produit de $a(a+1)$ où $a = 10000!$, $a = 15000!$, etc. . Même question pour des polynômes en une variable (à générer par exemple avec `symb2poly(randpoly(n))` ou avec `poly1[op(ranm(.))]`).
4. Programmer un algorithme de multiplication des polynômes par l'algorithme de Karatsuba. Observe-t-on la complexité attendue ? Si ce n'est pas le cas, essayez d'expliquer pourquoi (par exemple non utilisation de l'instruction d'affectation en place).
5. Comparer le temps de calcul de $a^n \pmod m$ par la fonction `powmod` et la méthode prendre le reste modulo $m$ après avoir calculé $a^n$.
   Programmez la méthode rapide et la méthode lente. Refaites la comparaison. Pour la méthode rapide, programmer aussi la version itérative utilisant la décomposition en base 2 de l'exposant : on stocke dans une variable locale $b$ les puissances successives $a^{2^0} \pmod m, a^{2^1} \pmod m, ..., a^{2^k} \pmod m, ...$, on forme $a^n \pmod n$ en prenant le produit modulo $m$ de ces puissances successives lorsque le bit correspondant est à 1 (ce qui se détecte par le reste de divisions euclidiennes sucessives par 2, le calcul de $b$ et du bit correspondant se font dans une même boucle).
6. Déterminer un entier $c$ tel que $c = 1 \pmod 3$, $c = 3 \pmod 5$, $c = 5 \pmod 7$ et $c = 2 \pmod{11}$.
7. Calculez dans $\mathbb{Z}/11\mathbb{Z}$
$$\prod_{a=0}^{10}(x-a)$$
8. Construire un corps fini de cardinal 128 (`GF`), puis factoriser le polynôme $x^2 - y$ où $y$ est un élément quelconque du corps fini. Comparer avec la valeur de $\sqrt{y}$.
9. Utiliser la commande `type` ou `whattype` ou équivalent pour déterminer la représentation utilisée par le logiciel pour représenter une fraction, un nombre complexe, un flottant en précision machine, un flottant avec 100 décimales, la variable $x$, l'expression $\sin(x)+2$, la fonction `x->sin(x)`, une liste, une séquence, un vecteur, une matrice. Essayez d'accéder aux parties de l'objet pour les objets composites (en utilisant `op` par exemple).
10. Comparer le type de l'objet `t` si on effectue la commande `t[2]:=0;` après avoir purgé `t` ou après avoir affecté `t:=[1,2,3]` ?
11. Comparer l'effet de l'affectation dans une liste et dans un vecteur ou une matrice sur votre logiciel (en Xcas, on peut utiliser =< au lieu de := pour stocker par référence).
12. Voici un programme qui calcule la base utilisée pour représenter les flottants.

```
Base():={
  local A,B;
  A:=1.0; B:=1.0;
  while (evalf(evalf(A+1.0)-A)-1.0=0.0) { A:=2*A;};
  while (evalf(evalf(A+B)-A)-B<>0) { B:=B+1;}
  return B;
} :;
```

Testez-le et expliquez.

13. Déterminer le plus grand réel positif $x$ de la forme $2^{-n}$ ($n$ entier) tel que $(1.0 + x) - 1.0$ renvoie 0 sur PC avec la précision par défaut puis avec `Digits:=30`.

14. Calculer la valeur de $a := \exp(\pi\sqrt{163})$ avec 30 chiffres significatifs, puis sa partie fractionnaire. Proposez une commande permettant de décider si $a$ est un entier.

15. Déterminer la valeur et le signe de la fraction rationnelle

$$F(x,y) = \frac{1335}{4}y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + \frac{11}{2}y^8 + \frac{x}{2y}$$

en $x = 77617$ et $y = 33096$ en faisant deux calculs, l'un en mode approché et l'autre en mode exact. Que pensez-vous de ces résultats ? Combien de chiffres significatifs faut-il pour obtenir un résultat raisonnable en mode approché ?

16. Que se passe-t-il si on essaie d'appliquer l'algorithme de la puissance rapide pour calculer $(x+y+z+1)^k$ par exemple pour $k = 64$ ? Calculer le nombre de termes dans le développement de $(x + y + z + 1)^n$ et expliquez.

17. Programmation de la méthode de Horner
Il s'agit d'évaluer efficacement un polynôme

$$P(X) = a_nX^n + ... + a_0$$

en un point. On pose $b_0 = P(\alpha)$ et on écrit :

$$P(X) - b_0 = (X - \alpha)Q(X)$$

où :

$$Q(X) = b_nX^{n-1} + ... + b_2X + b_1$$

On calcule alors par ordre décroissant $b_n$, $b_{n-1}$, ..., $b_0$.

(a) Donner $b_n$ en fonction de $a_n$ puis pour $i \leq n - 1$, $b_i$ en fonction de $a_i$ et $b_{i+1}$. Indiquez le détail des calculs pour $P(X) = X^3 - 2X + 5$ et une valeur de $\alpha$ non nulle.

(b) Écrire un fonction `horn` effectuant ce calcul : on donnera en arguments le polynôme sous forme de la liste de ces coefficients (dans l'exemple `[1,0,-2,5]`) et la valeur de $\alpha$ et le programme renverra $P(\alpha)$. (On pourra aussi renvoyer les coefficients de $Q$).

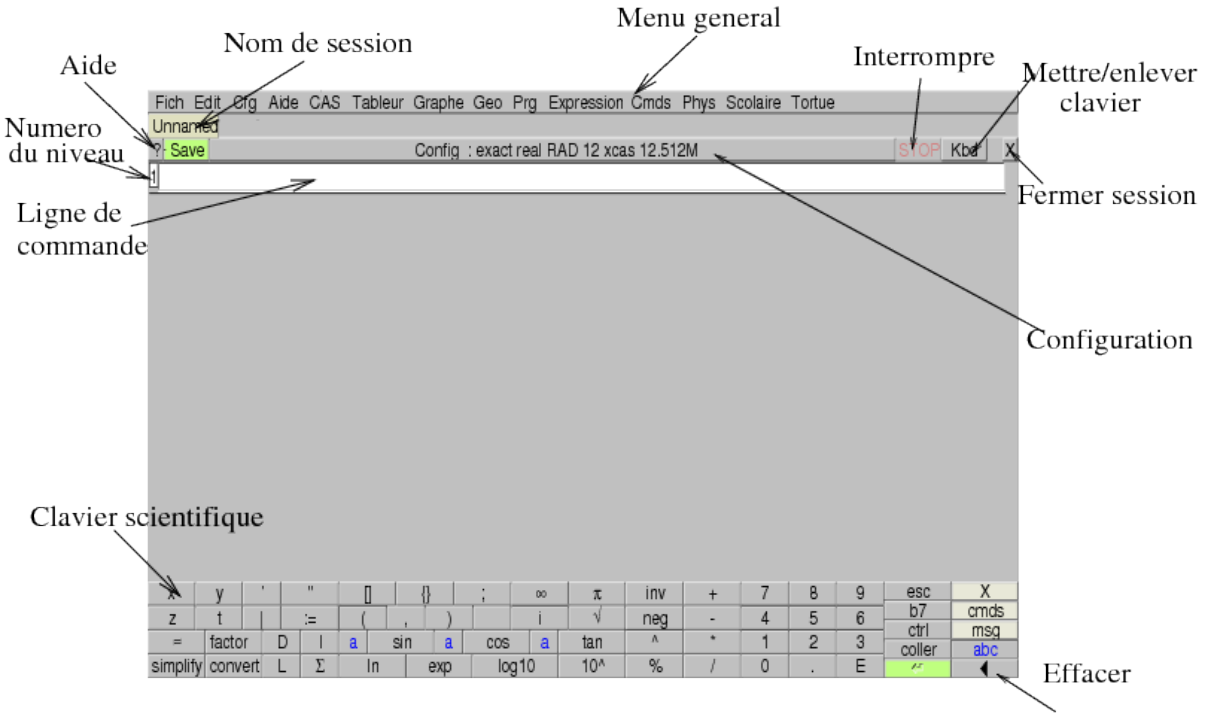
(c) En utilisant cette fonction, écrire une fonction qui calcule le développement de Taylor complet d'un polynôme en un point.

# 3 Premiers pas avec Xcas

Pour tester Xcas dans votre navigateur (sur PC/Mac, tablette ou smartphone), allez sur
`http://www-fourier.ujf-grenoble.fr/~parisse/xcasfr.html`
Pour télécharger Xcas, allez sur le site
`http://www-fourier.ujf-grenoble.fr/~parisse/install_fr.html`
Pour traiter les exemples, il est conseillé d'ouvrir Xcas :

- Sous Windows en installation locale, on clique sur l'icone `xcasfr` du bureau.
- Sous Linux, cherchez Xcas dans le menu Education. Si vous ne le trouvez pas, ouvrez un terminal et taper `xcas &`.
- sur Mac, cliquez sur Xcas dans le menu Applications du Finder.

Lors de la première utilisation, choisissez `Xcas` lorsqu'on vous demande de choisir une syntaxe, sauf si vous connaissez le langage Maple. L'interface apparaît comme suit au lancement de `Xcas`.




Vous pouvez la redimensionner. De haut en bas cette interface fait apparaître :

- un barre de menu gris cliquable : `Fich`, `Edit`, `Cfg`, `Aide`, `CAS`, `Tableur`, `Graphe`, `Geo`,...
- un onglet indiquant le nom de la session, ou `Unnamed` si la session n'a pas encore été sauvegardée,
- une zone de gestion de la session avec un bouton `?` pour ouvrir l'index de l'aide un bouton `Save` pour sauvegarder la session, un bouton `Config: exact real ...` affichant la configuration et permettant de la modifier, un bouton `STOP` permettant d'interrompre un calcul trop long, un bouton `Kbd` pour faire apparaitre un clavier ressemblant à celui d'une calculatrice, qui peut faciliter vos saisies, et un bouton `x` pour fermer la session
- une zone rectangulaire blanche numérotée 1 (première ligne de commande) dans laquelle vous pouvez taper votre première commande (cliquez si nécessaire pour faire apparaitre le curseur dans cette ligne de commande) : `1+1`, suivi de la touche "Entrée" ("Enter" ou "Return" selon les claviers). Le résultat apparaît au-dessous, et une nouvelle ligne de commande s'ouvre, numérotée 2.

Vous pouvez modifier l'aspect de l'interface et sauvegarder vos modifications pour les utilisations futures (menu `Cfg`).

Vous n'avez pour l'instant qu'à entrer des commandes dans les lignes de commandes successives. Si vous utilisez la version html de ce cours, vous pouvez copier-coller les commandes proposées depuis votre navigateur. Chaque ligne de commande saisie est exécutée par la touche "Entrée". Essayez par exemple d'exécuter les commandes suivantes.

```
1/3+1/4
sqrt(2)^5
solve(x+3=1,x)
50!
```

Toutes les commandes sont gardées en mémoire. Vous pouvez donc remonter dans l'historique de votre session pour modifier des commandes antérieures. Essayez par exemple de changer les commandes précédentes en :

```
1/3+3/4
sqrt(5)^2
solve(2*x+3=0)
500!
```

Notez que

- La saisie du signe `*` pour effectuer une multiplication est obligatoire (sauf entre un nombre et un nom de variable)
- pour effacer une ligne de commande, on clique dans le numéro de niveau à gauche de la ligne de commande, qui apparait alors en surbrillance. On appuie ensuite sur la touche d'effacement. On peut aussi déplacer une ligne de commande sélectionnée avec la souris.
- Le menu `Outils` regroupe les commandes les plus utilisées par thèmes (algèbre, analyse, probas...)
- Le menu `Edit` vous permet de préparer des sessions plus élaborées qu'une simple succession de commandes. Vous pouvez créer des groupes de lignes de commandes (sections), fusionner des niveaux en un seul niveau.
- Le menu `Prg` permet d'ouvrir un niveau éditeur de programmes et contient la plupart des instructions utiles pour programmer (cf. section 5).

Pour plus de détails sur l'interface, allez dans le menu Aide, Interface.

## 3.1 Aide en ligne

Les commandes de Xcas sont regroupées par thèmes dans les menus du bandeau gris supérieur : `Outils`, `Cmds`, `Prg`, `Graphe`,... Lorsqu'on sélectionne une commande dans un menu,

- soit l'index de l'aide s'ouvre à la commande sélectionnée (par exemple pour les commandes du menu `Cmds`). Cliquez sur le bouton `Details` pour afficher la page du manuel correspondant à la commande dans votre navigateur.
- soit une boite de dialogue s'ouvre vous permettant de spécifier les arguments de la commande (par exemple pour tracer une courbe depuis le menu `Graphe`)
- soit la commande est recopiée dans la ligne de commande. Pour connaitre la syntaxe de cette commande, appuyez sur le bouton `?` en haut à gauche, ou faites afficher la zone de `Messages` (en utilisant le menu `Cfg` puis montrer msg). Vous pouvez aussi configurer Xcas (menu `Cfg` puis `Configuration generale` puis cocher la case `Aide HTML automatique`) pour que la page correspondante du manuel s'ouvre automatiquement dans votre navigateur.

Le menu `Aide` contient les différentes formes d'aide possible : un guide de l'utilisateur (interface), un guide de référence (`Manuels->Calcul formel`, aide detaillée sur chaque commande), un `Index` (liste des commandes classées par ordre alphabétique avec une ligne d'entrée permettant de se déplacer facilement) et une recherche par mots clefs.

Si vous connaissez déjà le nom d'une commande et que vous désirez vérifier sa syntaxe (par exemple `factor`), vous pouvez saisir le début du nom de commande (disons `fact`) puis taper sur la touche de tabulation (située à

gauche de la touche A sur un clavier français) ou cliquer sur le bouton `?` en haut à gauche. L'index des commandes apparaît alors dans une fenêtre, positionné à la première complétion possible, avec une aide succinte sur chaque commande. Par exemple, vous voulez factoriser un polynôme, vous supposez que le nom de commande commence par `fact`, vous tapez donc `fact` puis la touche de tabulation, vous sélectionnez à la souris `factor` (ou un des exemples) puis vous cliquez sur OK.

Vous pouvez aussi saisir `?factor` pour avoir l'aide succinte en réponse. Si le nom que vous avez saisi n'est pas reconnu, des commandes proches vous sont suggérées.

# 4 Les objets du calcul formel

## 4.1 Les nombres

Les nombres peuvent être exacts ou approchés. Les nombres exacts sont les constantes prédéfinies, les entiers, les fractions d'entiers et plus généralement toute expression ne contenant que des entiers et des constantes, comme `sqrt(2)*e^(i*pi/3)`. Les nombres approchés sont notés avec la notation scientifique standard : partie entière suivie du point de séparation et partie fractionnaire éventuellement suivie de `e` et d'un exposant (comme `1e-8` ou `6.02e23`). Ainsi, `2` est un entier exact, `2.0` est la version approchée du même entier ; `1/2` est un rationnel, `0.5` est la version approchée du même rationnel. `Xcas` peut gérer des nombres entiers en précision arbitraire : essayez de taper `500!` et comptez le nombre de chiffres de la réponse.

On passe d'une valeur exacte à une valeur approchée par `evalf` ou `approx`, on transforme une valeur approchée en un rationnel exact par `exact` Les calculs sont effectués en mode exact si tous les nombres qui interviennent sont exacts. Ils sont effectués en mode approché si un des nombres est approché. Ainsi `1.5+1` renvoie un nombre approché alors que `3/2+1` renvoie un nombre exact.

```
sqrt(2)
evalf(sqrt(2))
sqrt(2)-evalf(sqrt(2))
exact(evalf(sqrt(2)))*10^9
exact(evalf(sqrt(2)*10^9))
```

Pour les nombres réels approchés, la précision par défaut est proche de 14 chiffres significatifs (la précision relative est de 53 ou 45 bits pour les réels flottants normalisés selon les versions de Xcas). Elle peut être augmentée, en donnant le nombre de décimales désiré comme second argument de `evalf`.

```
evalf(sqrt(2),50)
evalf(pi,100)
```

On peut aussi changer la précision par défaut pour tous les calculs en modifiant la variable `Digits`.

```
Digits:=50
evalf(pi)
evalf(exp(pi*sqrt(163)))
```

La lettre `i` est réservée à $\sqrt{-1}$ [1], il est déconseillé de la réaffecter par exemple pour l'utiliser comme indice de boucle.

```
(1+2*i)^2
(1+2*i)/(1-2*i)
e^(i*pi/3)
```

1. En mode maple, on utilise I

Xcas distingue l'infini non signé infinity ($\infty$), de +infinity ($+\infty$, raccourci inf) et de -infinity ($-\infty$).

```
1/0; (1/0)^2; -(1/0)^2
```

| Constantes prédéfinies | |
|---|---|
| pi | $\pi \simeq 3.14159265359$ |
| e | $e \simeq 2.71828182846$ |
| i | $i = \sqrt{-1}$ |
| infinity | $\infty$ |
| +infinity | $+\infty$ |
| -infinity | $-\infty$ |
| euler_gamma | $\gamma \simeq 0.577215664902$ |

## 4.2 Les caractères et les chaînes

Une chaîne est parenthésée par des guillemets ("). Un caractère est une chaîne ayant un seul élément.

```
s:="azertyuiop"
size(s)
s[0]+s[3]+s[size(s)-1]
concat(s[0],concat(s[3],s[size(s)-1]))
head(s)
tail(s)
mid(s,3,2)
l:=asc(s)
ss:=char(l)
string(123)
expr(123)
expr(0123)
```

| Chaînes | |
|---|---|
| asc | chaîne->liste des codes ASCII |
| char | liste des codes ASCII->chaîne |
| size | nombre de caractères |
| concat ou + | concaténation |
| mid | morceau de chaîne |
| head | premier caractère |
| tail | chaîne sans le premier caractère |
| string | nombre ou expression->chaîne |
| expr | chaîne->nombre (base 10 ou 8) ou expression |

## 4.3 Les variables

Dans Xcas, les variables sont identifiés par un nom de variable comportant un ou plusieurs caractères, ainsi ab désigne un nom de variable et pas le produit de 2 variables $a$ et $b$ qui s'écrit a*b. Xcas fait la différence entre majuscules et minuscules, la variable a et A ne sont pas identiques. On dit qu'une variable est formelle si elle ne contient aucune valeur : toutes les variables sont formelles tant qu'elles n'ont pas été affectées (à une valeur). L'affectation est notée :=. Au début de la session a est formelle, elle devient affectée après l'instruction a:=3, a sera alors remplacé par 3 dans tous les calculs qui suivent, et a+1 renverra 4. Xcas conserve le contenu des

variables de votre session, même si vous effacez la commande qui définit une variable. Si vous voulez que la variable `a` après l'avoir affectée, soit à nouveau une variable formelle, il faut la "vider" par une commande, ici `purge(a)`. Dans les exemples qui suivent, les variables utilisées sont supposées avoir été purgées avant chaque suite de commandes.

Il ne faut pas confondre

- le signe `:=` qui désigne l'affectation
- le signe == qui désigne une égalité booléenne : c'est une opération binaire qui retourne 1 ou 0 (1 pour true qui veut dire Vrai et 0 pour false qui veut dire Faux)
- le signe = utilisé pour définir une équation.

```
a==b
a:=b
a==b
solve(a=b,a)
solve(2*a=b+1,a)
```

On peut faire certains types d'hypothèses sur une variable avec la commande `assume`, par exemple `assume(a>2)`. Une hypothèse est une forme spéciale d'affectation, elle efface une éventuelle valeur précédemment affectée à la variable. Lors d'un calcul, la variable n'est pas remplacée mais l'hypothèse sera utilisée dans la mesure du possible, par exemple `abs(a)` renverra `a` si on fait l'hypothèse `a>2`.

```
sqrt(a^2)
assume(a<0)
sqrt(a^2)
assume(n,integer)
sin(n*pi)
```

La fonction `subst` permet de remplacer une variable dans une expression par un nombre ou une autre expression, sans affecter cette variable.

```
subst(a^2+1,a=1)
subst(a^2+1,a=sqrt(b-1))
a^2+1
```

**Remarque** : pour stocker une valeur dans une variable par référence, par exemple pour modifier une valeur dans une liste (un vecteur, une matrice), sans recréer une nouvelle liste mais en modifiant en place la liste existante, on utilise l'instruction =< au lieu de `:=`. Cette instruction est plus rapide que l'instruction `:=`, car elle économise le temps de copie de la liste. Cependant, son usage est déconseillé si on ne maitrise pas la différence entre référence et valeur.

## 4.4 Les expressions

### 4.4.1 Définition

Une expression est une combinaison de nombres et de variables reliés entre eux par des opérations : par exemple `x^2+2*x+c`.

Lorsqu'on valide une commande, `Xcas` remplace les variables par leur valeur si elles en ont une, et exécute les opérations.

```
(a-2)*x^2+a*x+1
a:=2
(a-2)*x^2+a*x+1
```

Certaines opérations de simplification sont exécutées automatiquement lors d'une évaluation :

- les opérations sur les entiers et sur les fractions, y compris la mise sous forme irréductible
- les simplifications triviales comme $x+0=x$, $x-x=0$, $x^1=x\ldots$
- quelques formes trigonométriques : $\cos(-x)=\cos(x)$, $\cos(\pi/4)=\sqrt{2}/2$, $\tan(\pi/4)=1\ldots$

Nous verrons dans la section 4.4.2 comment obtenir plus de simplifications.

### 4.4.2 Développer et simplifier une expression

En-dehors des règles de la section précédente, il n'y a pas de simplification systématique. Il y a deux raisons à cela. La première est que les simplifications non triviales sont parfois coûteuses en temps, et le choix d'en faire ou non est laissé à l'utilisateur ; la deuxième est qu'il y a en général plusieurs manières de simplifier une même expression, selon l'usage que l'on veut en faire. Les principales commandes pour transformer une expression sont les suivantes :

- `expand` : développe une expression en tenant compte uniquement de la distributivité de la multiplication sur l'addition et du développement des puissances entières.
- `normal` et `ratnormal` : d'un bon rapport temps d'exécution-simplification, elles écrivent une fraction rationnelle (rapport de deux polynômes) sous forme de fraction irréductible développée ; `normal` tient compte des nombres algébriques (par exemple `sqrt(2)`) mais pas `ratnormal`. Les deux ne tiennent pas compte des relations entre fonctions transcendantes (par exemple entre `sin` et `cos`).
- `factor` : un peu plus lente que les précédentes, elle écrit une fraction sous forme irréductible factorisée.
- `simplify` : elle essaie de se ramener à des variables algébriquement indépendantes avant d'appliquer `normal`. Ceci est plus coûteux en temps et "aveugle" (on ne contrôle pas les réécritures intermédiaires). Les simplifications faisant intervenir des extensions algébriques (par exemple des racines carrées) nécessitent parfois deux appels et/ou des hypothèses (`assume`) pour enlever des valeurs absolues avant d'obtenir la simplification souhaitée. Il est parfois plus judicieux d'appliquer une commande spécialisée, par exemple de réecriture trigonométrique (`texpand`, `tcollect`, ...) que `simplify`.
- `tsimplify` essaie de se ramener à des variables algébriquement indépendantes mais sans appliquer `normal` ensuite.

Dans le menu `Expression` du bandeau supérieur, les sous-menus de réécriture contiennent d'autres fonctions, pour des transformations plus ou moins spécialisées.

```
b:=sqrt(1-a^2)/sqrt(1-a)
ratnormal(b)
normal(b)
tsimplify(b)
simplify(b)
simplify(simplify(b))
assume(a<1)
simplify(b)
simplify(simplify(b))
```

La fonction `convert` permet de passer d'une expression à une autre équivalente, sous un format qui est spécifié par le deuxième argument.

```
convert(exp(i*x),sincos)
convert(1/(x^4-1),partfrac)
convert(series(sin(x),x=0,6),polynom)
```

| Transformations | |
|---|---|
| `simplify` | simplifier |
| `tsimplify` | simplifier (moins puissant) |
| `normal` | forme normale |
| `ratnormal` | forme normale (moins puissant) |
| `expand` | développer |
| `factor` | factoriser |
| `assume` | rajout d'hypothèses |
| `convert` | transformer en un format spécifié |

## 4.5 Les fonctions

### 4.5.1 Fonctions usuelles

De nombreuses fonctions sont déjà définies dans `Xcas`, en particulier les fonctions classiques. Les plus courantes figurent dans le tableau ci-après ; pour les autres, voir le menu `Cmds`.

| Fonctions classiques | |
|---|---|
| `abs` | valeur absolue |
| `round` | arrondi |
| `floor` | partie entière (plus grand entier $\leq$) |
| `ceil` | plus petit entier $\geq$ |
| `abs` | module |
| `arg` | argument |
| `conj` | conjugué |
| `im` | partie imaginaire |
| `re` | partie réelle |
| `sqrt` | racine carrée |
| `exp` | exponentielle |
| `log` | logarithme naturel |
| `ln` | logarithme naturel |
| `log10` | logarithme en base 10 |
| `sin` | sinus |
| `cos` | cosinus |
| `tan` | tangente |
| `asin` | arc sinus |
| `acos` | arc cosinus |
| `atan` | arc tangente |
| `sinh` | sinus hyperbolique |
| `cosh` | cosinus hyperbolique |
| `tanh` | tangente hyperbolique |
| `asinh` | argument sinus hyperbolique |
| `acosh` | argument cosinus hyperbolique |
| `atanh` | argument tangente hyperbolique |

### 4.5.2 Fonctions algébriques définies par l'utilisateur

Pour créer une nouvelle fonction, il faut la déclarer à l'aide d'une expression contenant la variable. Par exemple l'expression $x^2 - 1$ est définie par `x^2-1`. Pour la transformer en la fonction $f$ qui à $x$ associe $x^2 - 1$, trois possibilités existent :

```
f(x):= x^2-1
f:=x->x^2-1
f:=unapply(x^2-1,x)
f(2); f(a^2)
```

On peut définir des fonctions de plusieurs variables à valeurs dans $\mathbb{R}$ comme `f(x,y):=x+2*y` et des fonctions de plusieurs variables à valeurs dans $\mathbb{R}^p$ par exemple `f(x,y):=(x+2*y,x-y)`

### 4.5.3 Distinguer expression et fonction

Si `f` est une fonction d'une variable et `E` est une expression, `f(E)` est une autre expression. Il est essentiel de ne pas confondre fonction et expression. Si on définit : `E:=x^2-1`, alors la variable `E` contient l'expression $x^2-1$. Pour avoir la valeur de cette expression en $x=2$ il faut écrire `subst(E,x=2)` et non `E(2)` car `E` n'est pas une fonction. Lorsqu'on définit une fonction, le membre de droite de l'affectation n'est pas évalué. Ainsi l'écriture `E:=x^2-1; f(x):=E` définit la fonction $f : x \mapsto E$ car `E` n'est pas évalué. Par contre `E:= x^2-1; f:=unapply(E,x)` définit bien la fonction $f : x \mapsto x^2-1$ car `E` est évalué.

La fonction `diff` ou l'apostrophe postfixée `'` permet de calculer la dérivée d'une expression par rapport à une ou plusieurs de ses variables. Pour dériver une fonction $f$, on peut écrire `f'` ou `function_diff(f)`. Attention, $f(x)'$ ou `diff(f(x))` est une expression, pas une fonction.

```
E:=x^2-1
diff(E); E'
f:=unapply(E,x)
f'
diff(f(x))
f1:=function_diff(f)
```

Il ne **faut pas** définir la fonction dérivée par `f1(x):=diff(f(x))`, car `x` aurait dans cette définition deux sens incompatibles : d'une part la variable formelle de dérivation et d'autre part l'argument de la fonction `f1`. D'autre part, cette définition évaluerait `diff` à chaque appel de la fonction (car le membre de droite d'une affectation n'est jamais évalué), ce qui serait inefficace. Il faut donc utiliser `f1:=f'` ou `f1:=unapply(diff(f(x)),x)`.

### 4.5.4 Opérations sur les fonctions

On peut ajouter et multiplier des fonctions, par exemple `f:=sin*exp`. Pour composer des fonctions, on utilise l'opérateur `@` et pour composer plusieurs fois une fonction avec elle-même, on utilise l'opérateur `@@`.

```
f:=x->x^2-1
f1:=f@sin
f2:=f@f
f3:=f@@3
f1(a)
f2(a)
f3(a)
```

## 4.6 Listes, séquences, ensembles, vecteurs, matrices, polynômes

`Xcas` distingue plusieurs sortes de collections d'objets, séparés par des virgules :

- les listes (entre crochets)
- les séquences (éventuellement entre parenthèses)
- les ensembles (entre `set[` et `]`)

```
liste:=[1,2,4,2]
sequence:=(1,2,4,2)
ensemble:=set[1,2,4,2]
```

Les listes peuvent contenir des listes (c'est le cas des matrices), alors que les séquences sont plates (un élément d'une séquence ne peut pas être une séquence). Dans un ensemble, l'ordre n'a pas d'importance et chaque objet est unique. Il existe une autre structure, appelée table (ou annuaire ou dictionnaire), dont nous reparlerons plus loin.

Il suffit de mettre une séquence entre crochets pour en faire une liste ou entre `set[ ]` pour en faire un ensemble. On passe d'une liste à sa séquence associée par `op`, d'une séquence à sa liste associée en la mettant entre crochets ou avec la fonction `nop`. Le nombre d'éléments d'une liste est donné par `size` (ou `nops`).

```
se:=(1,2,4,2)
li:=[se]
op(li)
nop(se)
nops(se)
set[se]
size([se])
size(set[se])
```

Pour fabriquer une liste ou une séquence, on utilise des commandes d'itération comme `$` ou `seq` (qui itèrent une expression) ou `makelist` (qui définit une liste à l'aide d'une fonction).

```
1$5
k^2 $ (k=-2..2)
seq(k^2,k=-2..2)
seq(k^2,k,-2..2)
[k^2$(k=-2..2)]
seq(k^2,k,-2,2)
seq(k^2,k,-2,2,2)
makelist(x->x^2,-2,2)
seq(k^2,k,-2,2,2)
makelist(x->x^2,-2,2,2)
```

La séquence vide est notée `NULL`, la liste vide `[]`. Pour ajouter un élément à une séquence il suffit d'écrire la séquence et l'élément séparés par une virgule. Pour ajouter un élément à une liste on utilise `append`. On accède à un élément d'une liste ou d'une séquence en écrivant le nom de variable de la liste suivi par son indice mis entre parenthèses ou double-crochets (indices commencant à 1) ou entre simple crochets, le premier élément a alors comme indice 0 en mode Xcas (dans les autres modes les indices commencent à 1).

```
se:=NULL; se:=se,k^2$(k=-2..2); se:=se,1
li:=[1,2]; (li:=append(li,k^2))$(k=-2..2)
li[0],li[1],li[2]
li(1),li[[1]]
```

On peut modifier un élément d'une liste, séquence, ensemble en utilisant l'opérateur d'affectation `:=` précédé par le nom de variable de la liste indicié. Par exemple `li[1]:=4` modifie l'élément d'indice 1 de la liste contenue dans la variable `li`, les indices commencant à 0. De même pour `li(1):=5` et `li[[1]]:=5` mais en commencant les indices à 1. **Attention**, si vous utilisez l'indiciage commencant à 1 par `()`, il peut y avoir ambiguité entre modification d'une liste et définition d'une fonction, il est recommandé de lever l'ambiguite avec l'indiciage par `[[ ]]` commencant à 1 ou par `[ ]` commencant à 0.

Une copie de la liste modifiée est alors créée. Pour des raisons d'efficacité (temps de copie) on peut aussi utiliser l'opérateur d'affectation par référence (en place) =< mais attention cela modifiera toutes les variables pointant sur la liste originale. Par exemple `li[1]:=3` ou `li[1]=<4`.

Les vecteurs sont représentés par des listes, les matrices par des listes de listes de même longueur représentant les lignes de la matrice. Vous trouverez dans le menu `Cmds->Alglin` de nombreuses commandes pour manipuler vecteurs et matrices.

Les polynômes sont souvent définis par une expression, mais ils peuvent aussi être représentés par la liste de leurs coefficients par ordre de degré décroissant, avec comme délimiteurs `poly1[` et `]`. Il existe aussi une représentation pour les polynômes à plusieurs variables. Les fonctions `symb2poly` et `poly2symb` permettent de passer de la représentation expression à la représentation par liste et inversement, le deuxième argument détermine s'il s'agit de polynômes en une variable (on met le nom de la variable) ou de polynômes à plusieurs variables (on met la liste des variables).

| **Séquences et listes** | |
|---|---|
| `E$(k=n..m)` | créer une séquence |
| `seq(E,k=n..m)` | créer une séquence |
| `[E$(k=n..m)]` | créer une liste |
| `makelist(f,k,n,m,p)` | créer une liste |
| `op(li)` | passer de liste à séquence |
| `nop(se)` | passer de séquence à liste |
| `nops(li)` | nombre d'éléments |
| `size(li)` | nombre d'éléments |
| `sum` | somme des éléments |
| `product` | produit des éléments |
| `.+ .- .* ./ .^` | opération élément par élément |
| `cumSum` | sommes cumulées des éléments |
| `apply(f,li)` | appliquer une fonction à une liste |
| `map(li,f)` | appliquer une fonction à une liste |
| `poly2symb` | polynôme associé à une liste |
| `symb2poly` | coefficients d'un polynôme |

## 4.7 Les tables

Il s'agit de conteneurs associatifs permettant de créer des tableaux indiciés par n'importe quel type d'indice ordonné et non d'un indice entier compris entre 0 et la taille de la table. C'est typiquement ce qu'on utilise pour un annuaire. Par exemple

`tel["gaston"]:=123456789; tel["lagaffe"]:=987654321`

crée une table contenant deux entrées dont les indices sont des chaines de caractères.

Attention, c'est ce type de conteneur qui est créé par défaut lorsqu'on écrit une affectation comme `t[0]:=5` lorsque `t` n'a pas été initialisé à un type liste auparavant.

## 4.8 Instructions graphiques

Toutes les instructions du menu `Geo` ont un résultat graphique, par exemple `point(1,2)` affiche le point de coordonnées 1 et 2, `droite(A,B)` la droite passant par deux points $A$ et $B$ définis auparavant. On peut donner des attributs graphiques aux objets graphiques en ajoutant à la fin de l'instruction graphique l'argument `affichage=...` dont la saisie est facilitée par le menu `Graphic->Attributs`.

Lorsqu'une ligne de commande renvoie un objet graphique (ou une liste ou séquence terminant par un objet graphique), le résultat est affiché dans un repère 2-d ou 3-d selon la nature de l'objet généré. On peut controler le repère avec les boutons situés à droite du graphique, par exemple orthonormaliser avec le bouton _|_.

L'instruction `A:=click()` permet de définir une variable contenant l'affixe d'un point du plan que l'on clique avec la souris.

### 4.9 Temps de calcul, place mémoire

Le principal problème du calcul formel est la complexité des calculs intermédiaires. Elle se traduit à la fois par le temps nécessaire à l'exécution des commandes et par la place mémoire requise. Les algorithmes implémentés dans les fonctions de `Xcas` sont performants, mais ils ne peuvent pas être optimaux dans tous les cas. La fonction `time` permet de connaître le temps d'exécution d'une commande [2] (si ce temps est très court, `Xcas` exécute plusieurs fois la commande pour afficher un résultat plus précis). La mémoire utilisée apparaît dans les versions Unix dans la ligne d'état (en rouge à bas à gauche). Si le temps d'exécution d'une commande dépasse quelques secondes, il est possible que vous ayez commis une erreur de saisie. N'hésitez pas à interrompre l'exécution (bouton orange `stop` en bas à droite, il est alors conseillé de faire une sauvegarde de votre session, car certaines interruptions de calcul ne se font pas proprement et un crash peut survenir à la suite d'une interruption).

## 5 Programmation

Comme le texte définissant un programme ne tient en général pas sur une ou deux lignes, il est commode d'utiliser un éditeur de programmes. Pour cela, on utilise le menu `Prg->Nouveau programme` de Xcas. Les boutons assistants
`Fonctions, Test, Boucle` ou le menu `Prg->Ajouter` facilite la saisie des principales structures de controle de programmation. Xcas accepte plusieurs styles de syntaxe, style algorithmique en francais (celui des assistants par défaut), style proche du C, ou compatible maple, mupad, TI. On peut choisir un style particulier de programmation dans la configuration du CAS. Certaines constructions d'un style de programmation fonctionnent dans les autres styles, on s'efforcera toutefois de ne pas mélanger plusieurs styles dans un même programme...

### 5.1 Tests

On peut tester l'égalité de 2 expressions en utilisant l'instruction ==, alors que `!=` teste si 2 expressions ne sont pas égales. **Attention**, le test == n'effectue pas de réécriture d'expressions, par exemple pour savoir si `a` et `b` sont deux expressions mathématiquement équivalentes, testez si `simplify(a-b)==0`.

On peut aussi tester l'ordre entre 2 expressions avec <, <=, >, >=, il s'agit de l'ordre habituel sur les réels (pour des données numériques) ou de l'ordre lexicographique pour les chaines de caractères.

Un test renvoie 1 s'il est vrai, 0 s'il est faux. On peut combiner le résultat de deux tests au moyen des opérateurs logiques `and` ou `et` ou `&&` (et logique), `or` ou `ou` ou `||` (ou logique) et on peut calculer la négation logique d'un résultat de test par `not()` ou `non()` ou `!` (négation logique). On utilise ensuite souvent la valeur du test pour exécuter une instruction conditionnelle
`si alors sinon fsi;`
ou `if ... then ... else ... fi;`
ou en syntaxe compatible avec le langage C
`if (condition) { bloc_vrai } else { bloc_faux }`).

Par exemple, on pourrait stocker la valeur absolue d'un réel $x$ dans $y$ par :
`si x>0 alors y:=x; sinon y:=-x; fsi;`
(on peut bien sur utiliser directement `y:=abs(x)`).

2. Sous Linux, deux temps de calcul sont renvoyés, le temps mis par le processeur pour effectuer le calcul sans tenir compte des autres taches effectuées en même temps, et le temps réel tel qu'il apparait si on chronomètre. Le temps réel est supérieur ou égal au temps processeur, sauf si on dispose de plusieurs coeurs et que Xcas sait exploiter ces coeurs pour faire des calculs en parallèle.

## 5.2 Boucles

On peut exécuter des instructions plusieurs fois de suite en utilisant une boucle définie (le nombre d'exécutions est fixé au début) ou indéfinie (le nombre d'exécutions n'est pas connu). On utilise en général une variable de controle (indice de boucle ou variable de terminaison).

— Boucle définie

`pour de jusque faire fpour;`

`for(init;condition;incrementation){ instructions }`

`for ... from ... to ... do ... od;`

Exemple, calcul de 10 !

```
f:=1; pour j de 1 jusque 10 faire f:=f*j; fpour;
f:=1; for (j:=1;j<=10;j++){ f:=f*j; }
```

**Attention** à ne pas utiliser `i` comme indice de boucle, car `i` est prédéfini (=$\sqrt{-1}$)

— Boucle indéfinie

`tantque ... faire ... ftantque;`

`while (...) { ... }`

Exemple, algorithme d'Euclide

```
tantque b!=0 faire r:=irem(a,b); a:=b; b:=r; ftantque;
```

**Attention**, si on utilise la syntaxe de type Maple `while ... do... od` à ne pas mettre le test entre parenthèses en style Xcas.

Xcas accepte aussi l'arrêt de boucle en cours d'exécution (`if (...) break;`) dont l'usage peut éviter l'utilisation de variables boolénnes de controle compliquées.

## 5.3 Fonctions (non algébriques)

La plupart des fonctions ne peuvent avoir une définition par une formule algébrique. On doit souvent calculer des données intermédiaires, faire des tests et des boucles. Il faut alors définir la fonction par une suite d'instructions, délimitées par `{ ... }`. La valeur calculée par la fonction est alors la valeur calculée par la dernière instruction ou peut être explicitée en utilisant le mot-clef `return` suivi de la valeur à renvoyer. L'exécution de `return` met un terme à la fonction même s'il y a encore des instructions après, ceci est souvent pratique (par exemple on peut quitter une fonction avec valeur de retour au milieu d'une boucle).

Pour éviter que les données intermédiaires n'interfèrent avec les variables de la session principale, on utilise un type spécial de variables, les variables locales, dont la valeur ne peut être modifiée ou accédée qu'à l'intérieur de la fonction. On utilise à cet effet le mot-clef `local` suivi par les noms des variables locales séparés par des virgules. Si une fonction calcule plusieurs données on peut les renvoyer dans une liste.

## 5.4 Exemple : le PGCD

Ajouter un niveau programme (menu Prg, nouveau programme).

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

On clique ensuite sur le bouton OK, si tout va bien, le programme `pgcd` est défini et on peut le tester dans une ligne de commande par exemple par `pgcd(25,15)`. S'il y a des erreurs, la première erreur rencontrée est affichée et la position de l'erreur est mise en surbrillance dans l'éditeur de programmes.

## 5.5 Exécuter en pas à pas et mettre au point

La commande `debug` permet de lancer un programme en mode d'exécution pas à pas, ce qui peut servir à comprendre le déroulement d'un algorithme, mais aussi à corriger un programme erroné. `debug` ouvre une fenêtre permettant de diriger l'exécution du programme passé en argument. Par exemple, on entre le programme :

```
carres(n):={
  local j,k;
  k:=0;
  pour j de 1 jusque n faire
    k:=k+j^2;
  fpour;
  return k;
};:;
```

On tape pour debugger le programme `carres` ci-dessus :
`debug(carres(5))`
cela ouvre la fenêtre du debugger. En cliquant sur le bouton `sst` (raccourci F5) on peut exécuter pas à pas le programme en visualisant l'évolution des valeurs des variables locales et des paramètres du programme. Cela permet de détecter la grande majorité des erreurs qui font qu'un programme ne fait pas ce que l'on souhaite. Pour des programmes plus longs, le debugger permet de controler assez finement l'exécution du programme en placant par exemple des points d'arrêt.

Exercice : exécuter en mode pas à pas le programme `pgcd` pour quelques valeurs des arguments.