Introduction au calcul formel avec Maple

(T.P. du module LMB3 — licence de mathématiques)

Maximilian F. Hasler (mhasler@univ-ag.fr)

Département Scientifique Interfacultaire de l'Université Antilles–Guyane,
Campus de Schoelcher, B.P. 7209, 97275 Schoelcher cedex

septembre 2002

Table des matières

Préface

Ce document a été distribué aux étudiants de la licence de mathématiques, enseignée à Schoelcher au premier semestre 2002/2003, comme documentation complémentaire pour les TP du module LMB3, « calcul scientifique et programmation linéaire ».

Les TP constituent une initiation à l'utilisation d'un système de calcul formel, en l'occurrence le logiciel Maple ou MuPAD. La syntaxe de ces deux langages (définition de variables et fonctions, nom des fonctions de base du système) est presque la même. Nous les avons choisis pour les raisons suivantes :

- Maple est un des logiciels de calcul formel les plus puissantes du marché, et aussi un des plus répandues. D'autre part, c'est l'un des logiciels autorisés dans des concours de l'éducation nationale, par exemple dans l'agrégation de mathématiques (option calcul scientifique).
- L'étudiant qui apprend à se servir de Maple, peut très facilement s'adapter à MuPAD. MuPAD a l'avantage d'être disponible gratuitement, en tout cas pour une utilisation dans l'enseignement, sous certaines conditions.

  On peut directement télécharger par internet ce logiciel pour différents systèmes d'exploitation, notamment pour linux qui est lui-même gratuitement disponible, très fiable et immune contre les virus.
- Ces logiciels ont la plupart des procédures écrites dans le langage lui-même, ce qui permet de decouvrir le fonctionnement du système en affichant la définition de ces fonctions.

  D'autre part il y a une aide en-ligne très conviviale qui permet de parcourir les chapitres du manuel en fonction du sujet, mais aussi une recherche de mots clés qui ne sont pas forcément le nom exact de la fonction recherchée (que l'on ne connaît souvent pas).
- Ils utilisent une notation très naturelle, par exemple "`f := x -> x*sin(x)`" définit la fonction $f : x \mapsto x \sin x$ (cette même fonction peut se noter aussi "`(x->x)*sin`" ou encore "`proc(x) x*sin(x) end`"); "`f', f'', f@g`" sont respectivement la première et seconde dérivée de f (dans MuPAD) et la composée $f \circ g$, etc.

Le présent document est distribué, parmi d'autres, pour essayer de limiter au minimum les difficultés « techniques » qui sont toujours à l'ordre du jour lors d'un contact humain–ordinateur, afin que l'étudiant puisse consacrer le maximum de ses capacités aux aspects mathématiques du sujet.

Schoelcher, septembre 2002

Maximilian F. Hasler

1 Introduction au calcul formel

1.1 Introduction

Par un système de calcul formel on entend un logiciel permettant d'effectuer des calculs non seulement numériques, mais également analytiques. cela signifie que les résultats ne sont pas que des nombres (entiers, reels, complexes ; resp. leur représentation machinelle dite à virgule flottante), mais des expressions mathématiques quelconques, comportant notamment aussi des inconnues.

Exemple : calcul de $\sum_{n=1}^{\infty} \frac{1}{n^2}$. Avec une calculette où logiciel de calcul numérique, on n'obtient qu'une approximation. Par exemple, si la précision est de 5 chiffres, les premiers 250 termes vont donner 1.64493, mais les termes qui s'ajouteront seront négligés car $(1/250)^2$ est moins qu'un $1/100\,000^e$ du résultat.

Par contre, dans un logiciel tel que Maple, MuPAD, Mathematica, Derive, Reduce,... on tape :

```
> sum( 1/n^2 , n=1..infinity ) ; #  syntaxe Maple et MuPAD
```

$$\frac{\pi^2}{6}$$

c-à-d. on obtient le résultat exact sous forme symbolique. Pour en afficher la valeur numérique avec une précision demandée, il suffit de faire

```
> Digits := 50 ; evalf( %% ) ; #  %% = avant-dernier résultat (Maple)
• DIGITS := 50 ; float( %2 ) ; #  %2 = avant-dernier résultat (MuPAD)
```

1.6449340668482264364724151666460251892189499012068

Le calcul de la somme $\sum_{n=1}^{\infty} \frac{1}{n}$ est encore plus intéressant : sur une calculette ou logiciel de calcul numérique on obtient une valeur finie car dès que 1/n devient négligeable devant la somme $S_{n-1} = 1 + \cdots + \frac{1}{n-1}$ (au plus tard si $1/n$ est égal à la précision relative, par exemple 10^{-5}), la valeur de la somme ne change plus : $S_n = S_{n-1}$. On pourrait alors penser que S_n est une bonne approximation pour la limite (somme de la série). Cependant, si l'on effectue la sommation en commençant par les plus petits termes, on obtient des valeurs beaucoup plus grands pour S_n, en prenant n assez grand. Car ainsi S_n est calculée presque exactement, jusqu'à ce que le dernier terme, 1, devient négligeable devant S_n (qui croît très lentement : $S_{100\,000} < 12$!).
Dans un système de calcul formel, on obtient $= \infty$ comme valeur de la somme.

autres exemples :

```
> series( tan(x), x=a, 3 ) ;  # series = D.L. (Maple & MuPAD)
```

$$\tan a + (x-a)\left(\tan(a)^2+1\right) + (x-a)^2\left(\tan(a)+\tan(a)^3\right) + O((x-a)^3)$$

```
> int( 1/tan(x)^2, x) ;  # calcul de primitive (Maple & MuPAD)
```

$$-\frac{1}{\tan x} - x$$

J'espère qu'on voit l'intérêt de l'utilisation de tels outils. Comment les utiliser, c'est l'objet de la premiere partie de ce cours ; comment cela fonctionne, ce sera la 2^e partie.

1.2 Types de données et opérateurs

1.2.1 Types élémentaires

Les types de données de base sont les "literaux" (*literal* en anglais et dans Maple) : entiers *(integer)*, fractions *(fraction)*, « flottants » *(float)* et *string*s.

Les *entiers* ont une taille quasiment arbitraire (limitée en MuPAD uniquement par la memoire disponible ; en Maple à 0.5×10^6 ou 30×10^9 chiffres selon la machine (32 ou 64 bit).

Les *fractions* sont définis au moyen de 2 entiers sans diviseur commun, dont le 2e est strictement positif.

Les '*float*' (nombres a virgule flottante) sont représentés comme 2 entiers qui sont la *mantisse* et l'*exposant*, leur valeur est $mantisse \times 10^{\text{exposant}}$. (La taille de l'exposant étant généralement plus limité ; e.g. $exp < 2^{31} \approx 2 \times 10^9$.) Ces *float* ne sont pas à confondre avec les 'hardware float' utilisés par le processeur de la machine, qui ont un domaine et une précision beaucoup plus limité : typiquement, 16 décimales et une valeur inférieure à 10^{308} (11 bit d'exposant (-1024..1023) et 53 bit de mantisse ; cela signifie que $1 + 2^{-53} = 1$ et $2^{-1024} = 0$) — c'est généralement la précision de tout autre logiciel calculateur : calculette Windows etc., logiciels de calcul numérique tels que SciLab, MatLab,..., langages de programmation (C,...).
Exercice : essayer sur différentes "machines", et/ou logiciels à partir de quel $n \in \mathbb{N}$, le calcul $1 + 10^{-n} - 1$ (puis 10^{-n}) donne un résultat égal à zéro.

Enfin, les *strings* sont des chaines de charactères de la même longeur maximale que les entiers.

Mis à part ces types qui existent (sous forme moins générale) en quasiment tous les langages de programmation, il y a aussi celui des *noms* (*names*), qui correspondent aux inconnues en mathématiques, et auxquels on peut associer un objet quelconque comme leur valeur. N.B. : certains noms sont prédéfinies comme des constantes mathématiques : Pi, gamma..., d'autres comme fonctions (sin, cos, ...), et encore d'autres comme mots clés tels que "and", "or", "if", "then". Il est généralement déconseillé (parfois interdit) de les redéfinir autrement.

Les noms peuvent porter des *indices*, t.q. x_1, f_n,... qui sont à priori une expression quelconque. Ils sont généralement saisies à l'aide des crochets, par exemple $x[1]$, $f[n]$. Les indices peuvent aussi servir pour désigner les éléments d'objets composites (listes, matrices, tableaux,...), voir plus loin.

Les noms sans indices sont appellés *symboles*. L'ensemble des objets décrits jusqu'ici sont du type *atomique*, par opposition aux objets composites dont traite la suite.

1.2.2 Expressions

La plupart des expressions (terme général pour les objets que l'on manipule) sont formés à l'aide d'operateurs tel que $+$, $*$, etc. Les opérateurs ont une priorité, ce qui détermine par exemple que $A + B * C = A + (B * C)$ et non $(A + B) * C$. D'autre part, ils ont une associativité, ce qui fait que $A - B - C = (A - B) - C$ et non $A - (B - C)$. Quelques opérateurs sont donnés ci-dessous :

1.2.3 Les opérateurs classés par leur priorité.

(Attention, la définition de la priorité relative de certains opérateurs "exotiques" $(\cup, \cap, \mapsto, ...)$ pouvant varier d'un système à l'autre, il convient de mettre des parenthèses en cas de doute.)

1. `.` (associatif à gauche) : concaténation (noms et *strings*), ex. : a.b = ab
2. `%` (non-associatif) : rappel dernier résultat
3. `&`-operators (associatif à gauche) : à définir soi-même (Maple)
4. `!` (associatif à gauche) : factorielle
5. `^, **, @@` (non-associatif) : puissance et itérée de fonctions
6. `*, &*, /, @, intersect` (associatif à gauche) :
   `&*` = multiplicat^o non-commutative ; @ = composition de fct.
7. `+, -, union, minus` (associatif à gauche) : `minus` = soustraction d'ensembles
8. `mod` (non-associatif)
9. .. (non-associatif) : "range" = intervalle ou domaine (intégration, sommation, affichage (tracé de fonction), sélection d'éléments, etc.)
10. `<, <=, >, >=, =,<>` (non-associatif) : comparaison
11. `$` (non-associatif) : séquence, ex. : 1/n $ n=1..3 => 1, 1/2, 1/3
12. `not` (associatif à droite) : op. logique
13. `and` (associatif à gauche) : –//–
14. `or` (associatif à gauche) : –//–
15. `->` (associatif à droite) : déf. de fonction, ex. : `f := x -> x^2`
16. `,` (associatif à gauche) : séquence d'expression, ex. : $[a, b, c]$
17. `:=` (non-associatif) : définition (assigement) : `a :=5`.

Les opérateurs t.q. `^, **`, et `@@` sont définis non-associatifs et ainsi a^b^c n'est pas valable en Maple : il faut utiliser des parenthèses.

1.2.4 Ensembles, listes et tableaux

D'autres objets composites sont les listes $[a, b, c]$ et les ensembles : $\{1, 2, 3\}$. N.B. : $\{1, 2\} = \{2, 1\} = \{2, 1, 1\}$, mais $[1, 2] \neq [2, 1] \neq [2, 1, 1]$. Pour les ensembles, on peut utiliser les opérateurs

union, intersect, minus

par exemple, si A = {a,b} et B = {b,c}, alors :

A union B = {a,b,c}, A minus B = {a}, A intersect B = {b}.

A cause du fait que les ensembles ne sont jamais ordonnés de façon précise, on travaille plus souvent avec des listes. Pour accéder aux éléments d'une liste, on peut utiliser la sélection par moyen d'indice :

```
> L := [a,[b,c,d]] : L[ 2, 2..3] ;
```

$$[c, d]$$

Finalement, il y a les tableaux (*table*) qui sont créés implicitement des que l'on affecte une valeur à un nom indexé :

```
> t[1,1] := premier ;
```

va créér le tableau t qui contient l'entrée $(1, 1) = premier$. Si ensuite on fait

```
> t[deux] := second ;
```

on obtient le tableau t = *table*([(1,1) = premier, deux = second]). Un tableau peut avoir des indices quelconques. Un cas particulier est celui de l'*array*. Ici, les indices sont des entiers, dans un domaine précisé lors de la création :

```
> a := array( 0..2, -1..1 ) ;
```

crée un tableau à 2 dimensions, avec des indices $(i, j) \in \{0, 1, 2\} \times \{-1, 0, 1\}$. Les arrays du type $array(1..N)$ et $array(1..M, 1..N)$ correspondent aux vecteurs et matrices pour l'algèbre linéaire. On peut créer de tels objets en donnant la liste des valeurs, p.ex.

```
> v = array([x,y,z]), A = array([[a,b],[c,d]]) ;
```

$$v = [\, x \;\; y \;\; z \,], \quad A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

1.2.5 Fonctions et procédures

Un autre type d'objet composite est la *fonction*, plus précisément un *appel de fonction non évalué* ("unevaluated function call"). Celui-ci est de

la forme *nom*(*arguments*), où les arguments sont une suite (eventuellement vide) d'expressions (séparées par virgules).

Il faut distinguer ce type de *fonction* du type de *procédure*, qui correspond à une fonction proprement dite : dans cette terminologie, la fonction sin est du type *procédure*, alors que sin(1) ou sin(x) est du type *fonction*. Par exemple,
`> f := x -> x^(1/2) ;`
crée la procédure f qui n'est rien d'autre que la fonction racine carrée.

Mais les procédures et leur définition, c'est-à-dire la programmation, fait l'objet du chapitre suivant.

1.3 Procédures et programmation en Maple / MuPAD

1.3.1 Structures de contrôle

Les structures de contrôle permettent l'execution conditionnelle ou répétée d'une suite d'instructions. Il y a les formes suivantes :

a) execution conditionnelle :

`> if` *`expr`* `then` *`cmd1`* `[else` *`cmd2`*`] fi` (Maple)

• `if` *`expr`* `then` *`cmd1`* `[else` *`cmd2`*`] end_if` (MuPAD)

Si l'expression *expr* prend la valeur logique *vrai* (TRUE), alors les commandes *cmd1* sont executés, sinon les commandes *cmd2* (si présents, après *else*).

N.B. (mémotechnique) : en Maple, un bloc 'if' et 'do' se termine par 'fi' ou 'od', i.e. le mot-clé à l'envers.
En MuPAD, if, for, while, repeat, proc se terminent tous par end_if, end_for,..., end_proc.
N.B. : Dans des versions plus récentes de MuPAD, on peut aussi simplement utiliser `end` pour terminer toutes ces structures.

b) execution répétée. On distingue généralement 2 possibilités :

(i) boucle for :

`> for` *`name`* `from` *`init`* `to` *`end`* `[by` *`step`*`] do` *`cmd`* `od` (Maple)

• `for` *`name`* `from` *`init`* `to` *`end`* `[step` *`step`*`] do` *`cmd`* `end_for` (MuPAD)

ou :

`> for` *`name`* `in` *`expr`* `do` *`cmd`* `od` (Maple)

• `for` *`name`* `in` *`expr`* `do` *`cmd`* `end_for` (MuPAD)

Dans la 1^e forme, la variable *name* prend les valeurs de *init* à *end* avec un pas de *step* (si présent, sinon *step* = 1.— Si *end* < *init*, utiliser `downto` au lieu du `to`) ; chaque fois les commandes *cmd* sont executés.

Dans la 2^e forme, *name* prend comme valeur les éléments de *expr*, généralement une liste, un ensemble,... mais peut aussi être une somme, produit, etc. (cf. fonction `op()`).

Remarque 1.3.1 *La boucle for est caractérisée par le fait qu'on sait à l'avance le nombre (maximal) d'iterations.*

(ii) boucle while :

> `while` *cond* `do` *cmd* `od` (Maple)

• `while` *cond* `do` *cmd* `end_while` (MuPAD)

Tant que *cond* est vraie, executer *cmd*.

Remarque 1.3.2 *Etant donné qu'on ne sait pas quand la boucle va s'arrêter, il convient de s'assurer que la* cond *sera vérifiée un moment donné, ou de prévoir un autre moyen pour sortir de la boucle !*

Remarque 1.3.3 *En Maple, les 2 boucles utilisent la même structure qui s'écrit le plus généralement*

> `[for` *name*`]` `[from` *init*`]` `[to` *end*`]` `[by` *step*`]` `[in` *expr*`]`
> `[while` *cond*`]` `do` *cmd* `od`

Chacun des 5 éléments est optionnel, mais il faut au moins un `to` *ou* ***`while`****.*
Un ***`from`*** *ou* ***`by`*** *omis correspond à 1.*
Le ***`from...to...by...`*** *peut également être remplacé par '****`in`*** `expr`*'.*

Remarque 1.3.4 *En MuPAD existe la variante :*

• `repeat` *cmd* `until` *expr* `end_repeat ;`

qui execute cmd *jusqu'à ce que* expr *soit vraie.*

(iii) Sortie de boucle :

les commandes `break` et `next` (MuPAD) resp. `continue` (Maple) permettent d'interrompre une boucle (`break`) ou de passer à l'iteration suivante (`next` resp. `continue`).

1.3.2 Définition de fonctions et procédures

Une procédure est généralement définie à l'aide de la structure

`proc(` *arguments* `) begin` *commandes* `end_proc` (MuPAD)

`proc(` *arguments* `)` *commandes* `end` (Maple)

Le résultat est donné par la dernière valeur calculée dans la procédure. Par exemple,

```
> fac := proc( n ) if n <= 1 then 1 else n*fac(n-1) fi end ;     (Maple)
• fac := proc( n ) begin                                         (MuPAD)
      if n <= 1 then 1 else n*fac(n-1) end_if end_proc ;
```

définit une procédure (fonction) qui calcule la factorielle.

On peut sortir de la procédure avant d'arriver au `end_proc` à l'aide de la commande (fonction) `return( [resultat] )`, qui sort immédiatement de la procédure et "renvoie" le résultat donné. (En Maple, c'est `RETURN(...)`.) Quelque fois cela rend la lecture de la procédure beaucoup plus claire, notamment quand il y a beaucoup de `if...then...else` emboîtés.

Des procédures très simples tels que les fonctions usuellement rencontrées en mathematiques peuvent se définir aussi par l'opérateur flèche $->$:

```
> f := x -> x^(1/2) ;
```

est (presque) équivalent à

```
> f := proc(x) begin x^(1/2) end_proc ;
```

(ou encore à `f := sqrt`, fonction prédéfinie de racine carrée...)

> En effet, il y a ici des subtilités auxquels nous ne nous intéressons pas pour l'instant. En Maple, une procédure peut avoir des `options`, qui sont "operator,arrow" pour une procédure définie par `->`. "arrow" fait que la procédure est affichée avec la flèche, alors que "operator" la définit comme opérateur, ce qui a des conséquences pour la simplification etc. (Par exemple, une fonction constante `()-> cste` est automatiquement simplifiée en la constante `cste` simplement.)
>
> En MuPAD, il existe un type *fun* qui correspond à telles fonctions. On peut aussi les définir à l'aide de la fonction `func(`*`expression, variables`*`)`, qui correspond à la fonction `unapply(`*`expression, variables`*`)` en Maple.

Si on souhaite définir une fonction à plusieurs arguments, il faut les mettre entre parenthèses, par exemple :

```
> permut := (x,y) -> (y,x) ;
```

(Cette procédure à comme résultat une suite (séquence) de deux éléments.) Si on ne mettait pas de parenthèses, permut serait définie comme une suite de trois éléments : `x`, la fonction identité `y -> y`, et encore une fois `x`.

> Notons qu'on peut toujours donner plus d'arguments que prévu. Dans la procédure, on a access à tous ces arguments sous la forme de `args[1]`,...,`args[nargs]`, ou `nargs` est le nombre d'arguments fournis (en Maple). En MuPAD, il faut utiliser `args(i)` sous forme de fonction, et `args(0)` au lieu de `nargs`.
>
> La définition de la plupart des fonctions mathématiques commence par : `if nargs` $\neq$ `1 then ERROR(...)`.

Cependant, les fonctions définies par `->` ne peuvent contenir qu'un seul "calcul", c'est à dire une expression, mais pas de structures de contrôle tels

que boucles, etc. Pour faire des calculs plus complexes, il faut utiliser les procedures définies par `proc()...end`, qui peuvent contenir un nombre arbitraire de commandes (séparées par `;`), boucles, etc., et même des sous-procédures (locales).

On a aussi la possibilité de déclarer des *variables locales* (propres à la procédure) ou *globales* (communs à tous) à l'aide des instructions `local` *[variables]* `;` `global` *[variables]* `;` directement après `proc()`.

> On n'a pas la place ici pour parler du "lexical scoping" que fait MuPAD et Maple : cela signifie qu'une procédure 'herite' les variables d'une autre qui fait appel à cette procédure, et il y a toute une hierarchie qui détérmine qui connait quelle variable et qui est sensible à une redéfinition d'une variable (du même nom) ailleurs.

Finalement, nous remarquons qu'on se limitera pour la plupart des TD à l'utilisation des boucles en interactive et que nous n'utiliserons guère de procédures.

1.3.3 fonctions élémentaires prédéfinies en Maple et MuPAD

Fonctions numériques. Bien sûr que ces logiciels connaissent déjà un grand nombre de fonctions mathématiques, tels que

- les fonctions trigonometriques, hyperboliques, et leurs inverses :
  sin, cos, tan, sec (=1/cos), csc (=1/sin), cot, sinh, cosh, tanh, sech=1/cosh, csch=1/sinh, coth ;
  arcsin, arccos, arctan, arcsec, arccsc, arccot, arcsinh, arccosh, arctanh, arcsech, arccsch, arccoth
- abs - absolute value of real or complex number
- argument - argument of a complex number
- bernoulli - Bernoulli numbers and polynomials
- Beta - Beta function
- binomial - binomial coefficients
- ceil - smallest integer greater than or equal to a number
- Chi - hyperbolic cosine integral
- Ci - cosine integral
- conjugate - conjugate of a complex number or expression
- csgn - complex "half-plane" signum function
- dilog - dilogarithm function
- Dirac - Dirac delta function
- Ei - exponential integrals

– erf - error function
– erfc - complementary error function and its iterated integrals
– erfi - imaginary error function
– euler - Euler numbers and polynomials
– exp - exponential function
– factorial - factorial function
– floor - greatest integer less than or equal to a number
– frac - fractional part of a number
– GAMMA - Gamma and incomplete Gamma functions
– GaussAGM - Gauss arithmetic geometric mean
– HankelH1, HankelH2 - Hankel functions (Bessel functions of the 3rd kind)
– harmonic - partial sum of the harmonic series
– Heaviside - Heaviside step function
– hypergeom - generalized hypergeometric function
– ilog10, ilog - integer logarithms
– Im - imaginary part of a complex number
– Li - logarithmic integral
– ln - natural logarithm (logarithm with base exp(1) = 2.71...)
– lnGAMMA - log-Gamma function
– log - logarithm to arbitrary base
– log10 - log to the base 10
– max, min - maximum/minimum of a sequence of real values
– pochhammer - pochhammer symbol
– polar - polar representation of complex numbers
– polylog - polylogarithm function
– Psi - polygamma function
– Re - real part of a complex number
– round - nearest integer to a number
– signum - sign of a real or complex number
– Shi - hyperbolic sine integral
– Si - sine integral
– Ssi - shifted sine integral
– sqrt - square root
– surd - non-principal root function
– trunc - nearest integer to a number in the direction of 0
– Zeta - Riemann and Hurwitz zeta functions

...et bien d'autres fonctions spéciales numériques : fonctions d'Airy, Anger, Bessel, Elliptiques, Fresnel, Jacobi, Kelvin, Kummer, Legendre, Lerch, Lommel, Meijer, Struve, Lambert, Weber, Weierstrass, Whittaker...

Cette enorme bibliothèque transforme Maple ou MuPAD en un véritable dictionnaire de mathématiques : il suffit de faire '`?fonction`' pour avoir la définition et certaines propiétés de toutes ces fonctions.

Fonctions de calcul formel. Mis à part de ces fonctions "arithmétiques" il en existent bien d'autres qui concernent la manipulation des expressions, et qui nous seront au moins aussi utiles. Commençons par

`diff(expr, x)` : dérivation de l'*expression* par rapport aux variables x (peut être une séquence).

`D(f)` : dérivation de la *fonction* f ; $D[i](f)$ dérive par rapport aux i^{iemes} variables ($[i]$ peut être une liste d'entiers positifs).

`evalf(x[,n])` : Approximation en nombres flottantes de x (à n chiffres précis). Pour changer la précision, on peut modifier la variable *Digits* en Maple. (`float(x)` et *DIGITS* en MuPAD) :

`int(expr, variable[=a..b])` : primitive ou intégrale définie.

`sum(expr, variable[=a..b])` : somme.
(utiliser `add(f(v), v=a..b)` pour addition explicite)

`prod(expr, variable[=a..b])` : produit. (`mul(...,v=a..b)` pour multiplication explicite).

`simplify(expr [, relations])` : "simplification", en utilisant les relations donnés. (Ne fait pas toujours ce qu'on veut... Notamment, met tout sur un denominateur commun, dès qu'il y a un quotient dans une somme.)

`factor(expr)` : factorise (fait généralement ce qu'on veut). Pour factoriser des nombres, utiliser `ifactor()` dans Maple. (`Factor()` en MuPAD version 1.x.)

`expand(expr)` : le contraire : transforme tout en somme de produits etc.

`collect(expr, variables)` : 'collectionne' (met en facteur) les variables (liste).

`solve(equations, variables)` : résoud des équations par rapport aux variables (données comme ensembles {...} si plusieurs).
Utiliser *de préférence* dsolve, rsolve, fsolve dans le cas d'équa.diff., récurrence, ou approximation numérique (flottante) (bien que Maple essaie de trouver ce qui convient).
En MuPAD, une éq.diff. resp. récurrence est déclarée à l'aide des fonctions ode() ou rec(), par exemple :
- `solve( ode( y''(x)+4*y(x)=cos(x), y(x) ) );`

D'autre part, pour imposer une méthode, on peut appeler une fonction d'une bibliothèque spécifique, par exemple "numeric::solve" (approx. numérique) ou "linsolve" (pour syst. linéaire)

`plot(expr, var=a..b), plot(fct, a..b)` : tracer une fonction (en MuPAD, utiliser `plotfunc(...)`)

`series(expr, var=point [, ordre])` : D.L. (Taylor ou généralisé).

`limit(expr, var=point [, direction])` : limite.

Fonctions "techniques" Un autre ensemble de fonctions est de caractère plus "technique" que mathématique, mais néanmoins utile :

`help('fonction')` ou simplement `?fonction` : aide en-ligne (**très** conseillé, même (voire : surtout !) pour les fonctions déjà décrites ici.)

`op(i, expr)` : Donne le i^e opérand de l'expression, qui peut être une liste, somme, fonction...
Pour $i = a..b$, op(i,expr) = (op(j,expr) $ j=a..b) (suite des opérands).
Pour $i = [a, b]$, op(i,expr) = op(b,op(a,expr)) etc. (on déscend dans l'arborescence de l'expression).
(**N.B. :** en MuPAD, la syntaxe est : `op(expr, i)` !)

`nops(expr)` : nombre d'opérands

`map(fct, expr)` : applique la fonction fct à chaque opérand de l'expression *expr* (généralement une liste, mais pas nécessairement).
N.B. : c'est map(expr, fct) en MuAD.

`unapply( expr, var )` : convertir expr en une fonction des variables var. (en MuPAD 2.0 c'est fp::unapply(), avant c'était func().)

`C( expr [,...])` : convertir expr en code C. (generate::C() en MuPAD).

`latex( expr [,...]` : écrire expr en LaTeX. (generate::TeX() en MuPAD).