UNIVERSITÉ PARIS OUEST NANTERRE LA DÉFENSE
U.F.R. SEGMI — Année universitaire 2013 – 2014
Master d'économie — Cours de M. Desgraupes

Méthodes Numériques

Présentation de Maxima

Maxima est un outil dédié au calcul symbolique : il définit un langage interprété dont les commandes permettent d'effectuer des calculs algébriques. Ces commandes peuvent être exécutées directement une à une sur une ligne de commande ou peuvent être stockées dans des fichiers qui sont ensuite lus et exécutés par l'interpréteur.

Maxima est un logiciel libre et gratuit qui appartient à la famille des logiciels dits CAS (*Computer Algebra System*). Il est écrit en langage LISP.

Ce document est une présentation des fonctionnalités de base de Maxima pour une prise en main rapide de la syntaxe et des principales commandes.

1 Installation et démarrage

Le site officiel de Maxima est :

```
http://maxima.sourceforge.net
```

Pour se procurer Maxima, il suffit de télécharger une version binaire compilée. Il en existe pour toutes les plateformes usuelles (Unix, Mac OS X, Windows). On les trouvera à l'adresse suivante :

```
http://sourceforge.net/projects/maxima/files/
```

Sous Mac OS X et Windows, on démarre Maxima comme n'importe quelle application par un double-clic sur l'icone de l'application. Le programme affiche une fenêtre de terminal qui sert d'espace de travail, avec un préambule comme ceci :

```
Maxima 5.25.1 http://maxima.sourceforge.net
using Lisp SBCL 1.0.47
Distributed under the GNU Public License. See the file COPYING.
Dedicated to the memory of William Schelter.
The function bug_report() provides bug reporting information.
```

Le préambule est suivi d'une *invite* marquée par le préfixe `(%i1)`. C'est une ligne de commande. On saisit des instructions sur cette ligne et on les fait exécuter par Maxima en pressant la touche RETOUR (retour-chariot) ou la touche ENTER.

Depuis une fenêtre de terminal (sous Unix ou sous Mac OS X), il suffit de taper `maxima` pour lancer le programme. Si on ne veut pas afficher le préambule initial, il suffit d'exécuter la commande avec l'option *-q* comme ceci :

```
> maxima -q
```

2 Généralités

Chaque instruction de la ligne de commande doit être terminée par un point-virgule. Par exemple :

```
a : 10;
```

```
10
```

On peut faire plusieurs affectations sur une même ligne en les séparant par des virgules à l'intérieur d'une paire de parenthèses, comme ceci :

```
(b:4,c:5);
```

Si on ne souhaite pas que les résultats des opérations soient affichés, il suffit de terminer l'instruction par un signe dollar à la place du point-virgule. C'est utile lorsqu'une fonction exécute beaucoup de calculs intermédiaires et qu'on ne veut pas envahir l'écran inutilement.

```
a : 3 $
(b:4,c:5)$
```

2.1 Aide

Il y a plusieurs manières d'obtenir de l'aide concernant une fonction ou une variable.

• La fonction *apropos*

La fonction **apropos** renvoie une liste de toutes les fonctions ou variables dont le nom comporte une certaine chaîne de caractères. Par exemple :

```
(%i46) apropos("log");

(%o46) [%e_to_numlog,log,logabs,logarc,logconcoeffp,logcontract,
        logexpand,lognegint,lognumer,logout,logsimp,logx,logy,
        log_gamma,plog,superlogcon,taylor_logexpand,?%etolog]
```

• La fonction *describe*

La fonction **describe** permet d'afficher de l'aide concernant une fonction ou une variable particulière spécifiées en argument sous forme d'une chaîne de caractères. Par exemple (le résultat n'est pas reproduit ici) :

```
describe("log");
describe("lognegint");
```

On peut spécifier un second argument dans la fonction **describe** qui prend les valeurs *exact* ou *inexact*. Si l'argument est *inexact*, la fonction recherche l'aide pour tous les objets qui contiennent dans leur nom la chaîne spécifiée. S'il y a plusieurs possibilités, Maxima affiche une liste numérotée et attend que l'utilisateur précise un ou plusieurs nombres séparés par des espaces. Par exemple :

```
describe("log", inexact);

 0: Functions and Variables for Logarithms
 1: %e_to_numlog  (Functions and Variables for Logarithms)
 2: Base of natural logarithm  (Functions and Variables for Constants)
 3: cdf_logistic  (Functions and Variables for continuous distributions)
 etc...
```

On peut aussi utiliser un point d'interrogation simple ou double en début de ligne. C'est équivalent à la fonction **describe** avec l'argument *exact* ou *inexact* respectivement. Par exemple :

```
? log
?? log
```

Noter qu'il n'est pas nécessaire de terminer la ligne par un point-virgule dans ce cas mais il faut impérativement un espace entre le point d'interrogation et le terme qui suit (autrement le terme est interprété comme une variable LISP : voir à la section 12.1).

• La fonction *fundef*

La fonction **fundef** permet d'afficher la définition d'une fonction qui n'est pas définie comme primitive (*core command*, commande interne). Il faut que cette fonction soit connue de Maxima, autrement dit qu'elle ait été chargée en mémoire. Par exemple :

```
log10(x) := log(x) / log(10) $

(%i23) fundef(log10);

                                          log(x)
(%o23)                          log10(x) := -------
                                          log(10)
```

• La fonction *example*

On peut souvent obtenir des exemples variés d'utilisation d'une fonction au moyen de la fonction **example** :

```
(%i35) example(zeroequiv);


(%i36) zeroequiv(sin(2*x)-2*sin(x)*cos(x),x)

(%o36)                                true

(%i37) zeroequiv(x+%e^x,x)

(%o37)                                false

(%i38) zeroequiv(-log(b)-log(a)+log(a*b),a)

(%o38)                              dontknow
(%o38)                                done
```

Il n'existe pas d'exemples pour toutes les fonctions. La fonction **example** appelée sans arguments renvoie une liste des exemples existants.

2.2 Opérations de base

Les opérateurs arithmétiques de base sont les symboles + pour l'addition, − pour la soustraction, ⋆ pour la multiplication, / pour la division et ^ ou ⋆⋆ pour l'exponentiation. Le point '.' désigne la multiplication matricielle. On utilise des paires de parenthèses pour indiquer les précédences. Par exemple :

```
(%i1) (2^5 - 1) * 3;

(%o1)                                  93
```

Le quotient de deux nombres entiers est représenté par une fraction. Par exemple :

```
(%i2) (2^5 - 1) / 3;
```

$$(\%o2)\quad \frac{31}{3}$$

Pour que le calcul soit effectué en nombres réels, il faut qu'au moins un des nombres soit noté avec un point décimal. Par exemple :

```
(2^5 - 1) / 3.0;
```

$$10.33333333333333$$

On peut aussi utiliser l'une des fonctions **float** ou **bfloat**. Par exemple :

```
float(1/6);
```

$$.1666666666666667$$

• La fonction *bfloat*
La fonction **bfloat** permet d'avoir une précision arbitraire. La précision par défaut est de 16 chiffres après la virgule. Par exemple :

```
fpprec;
```

$$16$$

```
bfloat(1/7);
```

$$1.428571428571429b-1$$

On peut modifier la précision au moyen de la variable *fpprec* qui vaut 16 par défaut.

```
fpprec: 30;
```

$$30$$

```
bfloat(1/7);
```

$$1.428571428571428571428571428571b-1$$

2.3 Listes

On crée des listes dans Maxima au moyen d'une paire de crochets et en séparant les éléments par une virgule. Par exemple :

```
L : [a, b, c];
```

On extrait un élément d'une liste au moyen de son indice placé entre crochets (le premier terme correspond à l'indice 1) :

```
L[1];
```

```
                                    a
```

Lorsque des listes sont imbriquées, on utilise la fonction **part** pour extraire des éléments. Par exemple :

```
L: [[[- 1, 1], [1, 1]], [[[1, -3]], [[1, 2]]]];
part(L,2,2,1);
```

```
                                  [1, 2]
```

On reviendra sur la manipulation des listes à la section 3.1.

2.4 Étiquettes

Maxima permet de faire référence aux instructions précédentes ou à leur résultat au moyen des symboles $\%i$ (*i* pour *input*) ou $\%o$ (*o* pour *output*). Chaque commande est numérotée. On peut ainsi reprendre le résultat d'une opération précédente (ici la troisième que l'on désigne par `%o3`), comme ceci :

```
(%i5) %o3 * 12;

(%o5)                                124.0
```

Le symbole $\%$ tout seul représente par convention le résultat de la dernière opération. Par exemple :

```
(%i6) % - 2;

(%o6)                                122.0
```

Avec le symbole $\%th(n)$, on peut faire référence au résultat de la n-ième instruction qui précède la position actuelle. Par exemple, $\%th(2)$ est le résultat de l'avant-dernière instruction.

Remarque : les symboles `%i` et `%o` sont paramétrables. Ils sont définis au moyen des variables *inchar* et *outchar* dont on peut modifier la définition. Il existe aussi une variable *linenum* qui tient à jour le numéro de l'instruction courante.

2.5 Affichage

On peut contrôler la manière dont les expressions sont affichées au moyen de la variable booléenne *display2d* qui vaut TRUE par défaut. Une expression telle que $1/(1+x^2)$ sera affichée, dans ce cas, sous forme verticale comme ceci :

```
(%i2) 1/(1+x^2);

                                      1
(%o2)                           ------
                                 2
                                x  + 1
```

Si la variable *display2d* est fixée à la valeur FALSE, alors l'expression précédente sera affichée horizontalement sur une ligne :

```
(%i3) display2d : false $
(%i4) 1/(1+x^2);
```

```
(%o4) 1/(x^2+1)
```

• La fonction *box*

La fonction **box** permet d'entourer une expression d'une boîte consituée, par défaut, au moyen du symbole ". La syntaxe de cette fonction est :

```
box(expr)
box(expr, a)
```

Dans la deuxième forme, elle permet d'affecter une étiquette à la boîte. Par exemple :

```
(%i1) box((a+b)^2);

                                    """""""""""
                                    "         2"
(%o1)                               "(b + a)  "
                                    """""""""""

(%i2) box((a+b)^2,etqt);

                                    etqt""""""
                                    "         2"
(%o2)                               "(b + a)  "
                                    """""""""""
```

Le caractère utilisé pour dessiner la boîte peut être spécifié au moyen de l'option *boxchar*.

• La fonction *rembox*

La fonction **rembox** permet de retirer une boîte dessinée autour d'une expression ou d'une sous-expression. La syntaxe est :

```
rembox (expr, label)
rembox (expr)
```

• La fonction *display*

La fonction **display** affiche, au centre de la ligne, des équations dont le membre de gauche est le nom d'une expression et le membre de droite est sa valeur. Par exemple :

```
(%i1) f: x^2-x;

                                      2
(%o1)                                 x  - x

(%i2) x: 10;

(%o2)                                 10

(%i3) display(x,f);
```

```
                                    x = 10

                                       2
                                  f = x  - x

(%o3)                                done
```

• La fonction *ldisplay*

La fonction **ldisplay** fonctionne comme la fonction **display** mais attache une étiquette de la forme %tn à chaque équation qu'elle affiche. En reprenant l'exemple précédent, on obtient :

```
(%i5) ldisplay(x,f);

(%t5)                                x = 10

                                       2
(%t6)                             f = x  - x

(%o6)                             [%t5, %t6]
```

• La fonction *ldisp*

La fonction **ldisp** affiche sur la console une liste d'expressions en leur assignant une étiquette de la forme %tn qui permet de se référer à chacune d'entre elles dans les instructions suivantes. Par exemple :

```
(%i7) ldisp(p:(y+1)^2,q:expand(p));
                                                2
(%t7)                                    (y + 1)

                                        2
(%t8)                                  y  + 2 y + 1

(%o8)                                  [%t7, %t8]
```

• La fonction *dispterms*

La fonction **dispterms** permet d'afficher tous les termes d'une expression les uns en dessous des autres. L'opérateur est affiché en premier, suivi des sous-expressions. Par exemple :

```
(%i14) dispterms((a+b)^2*exp(-c^2));

*
        2
(b + a)

     2
  - c
%e

(%o14)                                 done
```

On utilise cette fonction lorsqu'une expression est trop longue.

• La fonction *print*

La fonction **print** affiche des expressions les unes à la suite des autres. Elle est utilisée en particulier pour créer des chaînes de caractères. Par exemple (ici x a la valeur 10) :

```
(%i18) print("La valeur de log(x) est", (ev(log(x),numer)))$

La valeur de log(x) est 2.302585092994046
```

Un espace est inséré entre chaque argument. Le résultat de cette commande est affiché même si l'instruction est terminée par un symbole dollar.

Pour afficher le contenu d'un fichier texte, on utilise la fonction **printfile** dont la syntaxe est :

```
(%i1)    printfile(path)
```

2.6 Affectations

L'affectation d'une valeur à une variable se fait au moyen du symbole deux-points. La valeur peut être un nombre ou une expression algébrique. Par exemple :

```
(%i5) S : %pi * r^2;

                                          2
(%o5)                                  %pi r

(%i6) r: 10;

(%o6)                                    10
```

Pour obtenir la valeur numérique approchée de la variable S, il faut ajouter le mot-clé *numer* :

```
(%i7) S, numer;

(%o7)                            314.1592653589793
```

Noter que le signe égal = n'effectue pas d'affectation. Il définit simplement une expression qui pourra être manipulée algébriquement par la suite. Par exemple :

```
(%i10) y=2;

(%o10)                                   y = 2

(%i11) y;

(%o11)                                     y
```

Dans cet exemple, y n'a pas reçu la valeur 2 : on a simplement écrit l'expression algébrique $y = 2$.

2.7 Variables

2.7.1 Variables prédéfinies

Maxima possède quelques variables prédéfinies :

- %pi : le nombre π ;
- %e : le nombre d'Euler, base des logarithmes népériens ;
- %phi : le nombre d'or $\frac{1+\sqrt{5}}{2}$;
- %i : le nombre imaginaire $\sqrt{-1}$;
- %gamma : la constante d'Euler-Mascheroni $\lim_{n\to\infty}\left(\sum_{k=1}^{n}\frac{1}{k}-\log(n)\right)$;
- inf : l'infini réel positif $+\infty$;
- minf : l'infini réel négatif $-\infty$;
- infinity : l'infini complexe.

2.7.2 Variables spéciales

La variable __ (double caractère de soulignement) désigne l'expression qui vient d'être entrée, autrement dit l'expression en cours d'évaluation.

La variable _ (simple caractère de soulignement) est la plus récente expression en entrée (e.g., `%i1`, `%i2`, `%i3`, ...).

On a vu déjà que la variable `%` désigne la plus récente expression en sortie (e.g., `%o1`, `%o2`, `%o3`, ...).

Dans des instructions composites (des blocs, des fonctions lambda, des expressions de la forme $(s_1, ..., s_n)$), le symbole `%%` représente la *valeur* de la dernière instruction.

Voici quelques exemples :

```
(%i1) print(__);

print(__)
(%o1)                                print(__)

(%i11) p: x^2*log(x);

                                      2
(%o11)                               x  log(x)

(%i12) _;

                                        2
(%o12)                           p : x  log(x)

(%i10) p: x^2+1;

                                      2
(%o10)                               x  + 1

(%i11) :lisp $_

((MSETQ) $P ((MPLUS) ((MEXPT) $X 2) 1))

(%i4) block (integrate (x^2, x), ev (%%, x=-1) - ev (%%, x=1));

                                       2
(%o4)                                - -
                                       3
```

2.8 Fonctions numériques

La table 1 de la page 13 contient une liste des principales fonctions mathématiques disponibles dans Maxima.

• La fonction *alias*

La fonction **alias** permet de renommer un ou plusieurs objets. Elle s'applique aux fonctions, aux variables ou aux tableaux. La syntaxe est :

```
alias (new_name_1, old_name_1,..., new_name_n, old_name_n)
```

2.9 Évaluation

Pour qu'une expression ne soit pas évaluée, on doit la faire précéder d'une simple apostrophe. Par exemple :

```
(%i9) diff (y, x);

(%o9)                                   0

(%i10) 'diff (y, x);

                                       dy
(%o10)                                 --
                                       dx
```

Pour forcer l'évaluation d'une expression au contraire, on utilise une double apostrophe. Par exemple, pour réexécuter une commande précédente, on utilise son étiquette en la faisant précéder de deux apostrophes. Par exemple :

```
''%i5
```

Il s'agit bien de deux apostrophes, et non pas d'un guillemet double.

Appliquée à une expression générale, la double apostrophe remplace l'expression par sa valeur. Appliquée à l'opérateur d'une expression, la double apostrophe change l'opérateur de prédicat en verbe.

• La fonction *ev*

La fonction **ev** permet d'évaluer des expressions dans certaines conditions. La syntaxe peut prendre deux formes synonymes :

```
ev ( expr, options );
expr, options ;
```

La deuxième forme n'est supportée qu'en mode interactif, c'est-à-dire en ligne de commande : on ne peut pas l'utiliser dans la définition d'une fonction ou dans un script.

Voici un exemple :

```
(%i6) ev( (a+b)^2, expand);

                                 2             2
(%o6)                           b  + 2 a b + a
```

abs	valeur absolue
acos	arc cosinus
acosh	arc cosinus hyperbolique
angle	argument d'un complexe
asin	arc sinus
asinh	arc sinus hyperbolique
atan	arc tangente
atan2(y,x)	arc tangente de y/x
atanh	arc tangente hyperbolique
ceil	plus petit entier supérieur
clock	heure et date
conj	conjugué d'un complexe
cos	cosinus
cosh	cosinus hyperbolique
cot	cotangente
csc	cosécante
cumsum	somme cumulée
date	date sous forme *jj-mm-aaaa*
exp	exponentielle
fix	arrondi à l'entier le plus près de 0
floor	plus petit entier inférieur
imag	partie imaginaire d'un complexe
length	longueur d'un vecteur
log10	logarithme décimal
log	logarithme népérien
max	maximum
mean	moyenne arithmétique
min	minimum
pow2	puissance de 2
prod	produit
rand	nombre au hasard entre 0 et 1
real	partie réelle d'un complexe
realmax	plus grand nombre réel sur la machine
realmin	plus petit nombre réel sur la machine
rem(m,n)	reste de la division de m par n
round	arrondi
sign	quotient $z/\|z\|$
sin	sinus
sinh	sinus hyperbolique

TABLE 1 – Fonctions mathématiques

On aurait pu écrire cet exemple plus simplement comme ceci :

```
(%i6) (a+b)^2,expand;
```

S'il y a plusieurs arguments, la syntaxe est la suivante :

```
ev (expr, arg_1, ..., arg_n)
expr, arg_1, ..., arg_n ;
```

Les arguments ou options peuvent aussi être des expressions de la forme `V=e` ou `V:e` qui sont évalués en parallèle. Par exemple :

```
(%i1) x+y, x = a+y;
(%o1)                         2 y + a
```

Les arguments peuvent aussi être des fonctions qui possèdent la propriété *evfun* (voir à la section 2.9). Par exemple

```
ev(expr, rectform, ratsimp)
```

est équivalent à

```
ratsimp ( rectform ( ev(expr) ) )
```

car les fonctions **rectform** et **ratsimp** ont la propriété *evfun*.

Certains arguments sont des mots-clés qui spécifient le mode d'évaluation. La documentation de la fonction **ev** mentionne les mots-clés suivants :

derivlist	*noeval*
detout	*nouns*
diff	*numer*
eval	*pred*
expand	*risch*
float	*simp*

• La propriété *evflag*

La propriété *evflag* concerne des variables booléennes. Si une variable x a cette propriété alors une expression telle que

```
(%i1)      expr, x
```

est équivalente à

```
(%i1)      ev(expr, x = true)
```

Par défaut, les variables suivantes ont la propriété *evflag* :

algebraic	*infeval*	*numer_pbranch*
cauchysum	*isolate_wrt_times*	*programmode*
demoivre	*keepfloat*	*radexpand*
dotscrules	*letrat*	*ratalgdenom*
%emode	*listarith*	*ratfac*
%enumer	*logabs*	*ratmx*
exponentialize	*logarc*	*ratsimpexpons*
exptisolate	*logexpand*	*simp*
factorflag	*lognegint*	*simpsum*
float	*lognumer*	*sumexpand*
halfangles	*m1pbranch*	*trigexpand*

À cette liste, on peut rajouter le mot-clé *numer* qui n'a pas la propriété *evflag* mais fonctionne de manière analogue. Voici quelques exemples :

```
(%i1)ev( log(3/4), float);

(%o1)                              - .2876820724517809

(%i8) cos(w*t), exponentialize;

                                 %i t w     - %i t w
                               %e       + %e
(%o8)                          ---------------------
                                         2
```

On peut attribuer la propriété *evflag* à une variable x au moyen de la fonction **declare** comme ceci :

```
(%i1)    declare(x, evflag)
```

• La propriété *evfun*

La propriété *evfun* concerne des fonctions. Si une fonction F a cette propriété alors une expression telle que

```
(%i1)     expr, F
```

est équivalente à

```
(%i1)     F ( ev (expr))
```

Plusieurs fonctions peuvent être composées. Ainsi, par exemple,

```
(%i1)    ev ( expr, F, G )
```

est équivalent à

```
(%i1)    G ( F ( ev(expr) ) )
```

Par défaut, les fonctions suivantes ont la propriété *evfun* :

bfloat	**ratexpand**
factor	**ratsimp**
fullratsimp	**rectform**
logcontract	**rootscontract**
polarform	**trigexpand**
radcan	**trigreduce**

À cette liste, on peut rajouter les mots-clés *expand*, *nouns*, *diff* et *integrate*. Par exemple :

```
(%i7) (a+b)*(c+d), expand;

(%o7)                          b d + a d + b c + a c
```

On peut attribuer la propriété *evfun* à une fonction F au moyen de la fonction **declare** comme ceci :

```
(%i1)     declare(F, evfun)
```

• La fonction *kill*

On utilise la fonction **kill** pour supprimer des variables ou des définitions de fonctions de la mémoire. La liste de toutes les variables en mémoire est contenue dans la variable *values*, la liste des fonctions est contenue dans la variable *functions*. Par exemple :

```
(%i1) values;

(%o1)                                  []

(%i2) aa: 123;

(%o2)                                  123

(%i3) values;

(%o3)                                 [aa]

(%i4) kill(aa);

(%o4)                                 done

(%i5) values;

(%o5)                                  []
```

On peut aussi utiliser la syntaxe globale suivante :

```
kill(all)
kill(allbut(x,y)).
```

2.10 Bases de numération

La variable *ibase* est une option qui permet de spécifier la base à utiliser pour la saisie en entrée de nombres entiers. La valeur par défaut est 10. On peut utiliser n'importe quelle valeur entre 2 et 36. Par exemple :

```
(%i1) ibase;

(%o1)                                  10

(%i2) ibase: 8;

(%o2)                                   8

(%i3) 64;

(%o3)                                  52

(%i4) 777;

(%o4)                                 511
```

```
(%i5) ibase: 12;

(%o5)                                    10
```

Noter que pour repasser en base 10 il a fallu donner la valeur 12 car on était en base 8 et que 12 est la valeur décimale de 10 dans cette base.

De manière analogue, il existe une variable *obase* qui définit la base de numération pour les valeurs entières produites en sortie par . Par exemple :

```
(%i5) obase: 16;

(%o5)                                    10

(%i6) 65536;

(%o6)                                   10000

(%i7) 65535;

(%o7)                                   0FFFF
```

Ici il faut comprendre que le fait de fixer la base à 16 renvoie la valeur 10 car il s'agit de 10 en base hexadécimale donc justement la valeur décimale 16 !

3 Expressions

3.1 Manipulation de listes

La liste est l'objet fondamental sur lequel reposent toutes les fonctionnalités de .

Création de listes

Une liste est une expression de la forme $[a, b, c, d]$: les éléments sont placés dans une paire de crochets et séparés par une virgule. Par exemple :

```
(%i2) L: [a,b,c,d];

(%o2)                                [a, b, c, d]
```

Les listes sont couramment imbriquées les unes dans les autres :

```
(%i3) LL: [L, [[e,f],g]];

(%o3)                       [[a, b, c, d], [[e, f], g]]
```

La fonction **length** renvoie la longueur d'une liste :

```
(%i4) length(L);

(%o4)                                    4

(%i5) length(LL);

(%o5)                                    2
```

• La fonction *makelist*

La fonction **makelist** permet de créer des listes génériques.On l'utilise de deux manières différentes illustrées par les exemples suivants :

```
(%i6) makelist(concat(x,i),i,1,6);

(%o6)                        [x1, x2, x3, x4, x5, x6]

(%i7) makelist(x=y,y,[a,b,c]);

(%o7)                          [x = a, x = b, x = c]

(%i8) makelist(x[y],y,L);

(%o8)                            [x , x , x , x ]
                                   a   b   c   d
```

La fonction **listp** teste si un objet est de type liste :

```
(%i58) listp(L);

(%o58)                                  true

(%i59) listp(cos);

(%o59)                                  false
```

Éléments d'une liste

Un élément d'une liste peut être désigné par son indice dans la liste placé entre crochets. Par exemple :

```
(%i13) L[3];

(%o13)                                    c

(%i14) LL[2][2];

(%o14)                                    g

(%i15) LL[2][1][2];

(%o15)                                    f
```

Pour tester si un élément fait partie d'une liste, on utilise la fonction **member** :

```
(%i66) member(d,L);

(%o66)                                  true
```

Les fonctions suivantes permettent d'extraire des éléments particuliers d'une liste :

first	**fourth**	**seventh**	**tenth**
second	**fifth**	**eighth**	**last**
third	**sixth**	**ninth**	

Par exemple :

```
(%i16) third(L);

(%o16)                                  c

(%i17) last(LL);

(%o17)                                [[e, f], g]
```

On peut manuellement ajouter des éléments en début ou en fin de liste au moyen des fonctions **cons** et **endcons**. Le premier argument est l'élément à ajouter et le deuxième est la liste elle-même. Par exemple :

```
(%i19) cons(x,L);

(%o19)                              [x, a, b, c, d]

(%i20) endcons(x,L);

(%o20)                              [a, b, c, d, x]
```

• La fonction *append*

La fonction **append** permet de concaténer les élements de plusieurs listes :

```
(%i27) append(L,L);

(%o27)                         [a, b, c, d, a, b, c, d]

(%i28) append(LL,[h,i,j]);

(%o28)                  [[a, b, c, d], [[e, f], g], h, i, j]
```

• La fonction *assoc*

La fonction **assoc** permet de rechercher des valeurs associées à des éléments dans des listes. Chaque élément de la liste doit être une expression ayant deux arguments : le premier élément sert de clé et le second sert de valeur associée. Par exemple :

```
(%i40) D: [x=1, u=2^3, [a,b]];

(%o40)                          [x = 1, u = 8, [a, b]]

(%i41) assoc(x,D);

(%o41)                                  1

(%i42) assoc(u,D);

(%o42)                                  8

(%i43) assoc(a,D);

(%o43)                                  b
```

• La fonction *atom*

La fonction **atom** indique si une élément d'une liste est atomique (nombre, chaîne, nom) ou composé. Elle renvoie une valeur logique. Par exemple :

```
(%i44) x: 1;

(%o44)                                  1

(%i45) atom(x);

(%o45)                                 true

(%i46) x: [1,2];

(%o46)                                [1, 2]

(%i47) atom(x);

(%o47)                                 false
```

Opérations sur les listes

La fonction **copylist** renvoie une nouvelle liste qui est la copie d'une liste existante.

La fonction **delete** supprime toutes les occurrences d'une sous-expression à l'intérieur d'une expression. Optionnellement, on peut spécifier le nombre d'occurrences à supprimer. Par exemple :

```
(%i50) delete(a, f(a,b,a,c,d,a));

(%o50)                              f(b, c, d)

(%i51) delete(a, f(a,b,a,c,d,a), 2);

(%o51)                             f(b, c, d, a)
```

La fonction **reverse** renvoie une liste qui renverse l'ordre des éléments d'une liste donnée :

```
(%i67) reverse(L);

(%o67)                             [d, c, b, a]
```

La fonction **rest** s'applique à une expression (et en particulier une liste) et renvoie une expression dont les n premiers éléments (si $n > 0$) ou les n derniers éléments (si $n < 0$) sont supprimés :

```
(%i69) rest(L,2);

(%o69)                                [c, d]

(%i70) rest(L,-2);

(%o70)                                [a, b]
```

3.2 Substitution

• La fonction *map*

La fonction **map** permet d'appliquer une fonction ou un opérateur aux termes d'une liste ou d'une expression. Il y a plusieurs façons de l'utiliser. Voici quelques exemples :

```
(%i10) map('log, [a,b,c]);

(%o10) [log(a),log(b),log(c)]

(%i11) map('log, a+b+c);

(%o11) log(c)+log(b)+log(a)

(%i12) map( "+", [a,b], [1,3]);

(%o12)                          [a + 1, b + 3]

(%i13) map('F, [a,b,c], [1,2,3], [r,s,t]);

(%o13)               [F(a, 1, r), F(b, 2, s), F(c, 3, t)]
```

Au lieu d'une fonction, on peut aussi spécifier une expression **lambda**.

• La fonction *fullmap*

La fonction **fullmap** applique le symbole spécifié à tous les éléments :

```
(%i12) map('log, a*cos(x)+b*sin(c*y));

(%o12) log(b*sin(c*y))+log(a*cos(x))

(%i13) fullmap('log, a*cos(x)+b*sin(c*y));

(%o13) log(b)*sin(log(c)*log(y))+log(a)*cos(log(x))
```

• La fonction *maplist*

La fonction **maplist** renvoie, sous forme d'une liste, la fonction appliquée à chaque partie des expressions passées en argument. La comparaison des deux exemples suivants montre la différence avec la fonction **map** :

```
(%i62) map(f,x+a*y+b*z);

(%o62)                       f(b z) + f(a y) + f(x)

(%i63) maplist(f,x+a*y+b*z);

(%o63)                       [f(b z), f(a y), f(x)]
```

• La fonction *scanmap*

La fonction **scanmap** applique une fonction de manière récurrente en parcourant les termes d'une expression. Par exemple :

```
(%i1) scanmap(f, ((x+a)^2+b)^c);

                                              f(2)          f(c)
(%o1)                f(f(f(f(f(x) + f(a))     ) + f(b))     )
```

• La fonction *outermap*

La fonction **outermap** applique une fonction au produit extérieur de plusieurs structures (liste, matrices). La syntaxe est :

```
outermap (f, a_1, ..., a_n)
```

où les arguments a_i peuvent être des listes, listes imbriquées, matrices etc. et où f est une fonction à n variables. Par exemple :

```
(%i4) f(x,y):=x^2-y^2;

                                        2    2
(%o4)                          f(x, y) := x  - y

(%i5) outermap(f, [a,b], [c,d]);

                       2    2   2    2     2    2   2    2
(%o5)                [[a  - c , a  - d ], [b  - c , b  - d ]]

(%i6) outermap(f, [a,b], matrix([r,s],[t,u]));

                   [  2    2   2    2 ]  [  2    2   2    2 ]
                   [ a  - r   a  - s  ]  [ b  - r   b  - s  ]
(%o6)             [[                   ], [                   ]]
                   [  2    2   2    2 ]  [  2    2   2    2 ]
                   [ a  - t   a  - u  ]  [ b  - t   b  - u  ]
```

• La fonction *apply*

La fonction **apply** opère uniquement sur une liste qui représente l'ensemble des arguments à passer à une fonction F. La syntaxe est :

```
apply (F, [x_1, ..., x_n])
```

Cela revient à évaluer $F(x_1, \ldots, x_n)$. Par exemple :

```
(%i7) apply(max,[-3,2,8,-5,6]);

(%o7)                                  8
```

• La fonction *subst*

La fonction **subst** a la syntaxe suivante :

```
subst (a, b, c)
subst (eq_1, expr)
subst ([eq_1, ..., eq_k], expr)
```

Elle substitue a à la place de b dans l'expressions c. Il faut que b soit un atome ou une sous-expression complete. Par exemple :

```
(%i6) subst(x+y,z,a*z^2+b*z+c);
                                        2
(%o6)                        a (y + x)  + b (y + x) + c
```

Si on spécifie une ou plusieurs équations, elles servent à effectuer la substitution dans l'expression. Elles sont appliquées séquentiellement. Par exemple :

```
(%i9) subst([a*x=b, b=c], a*x+c);

(%o9)                                      2 c
```

Les fonctions **sublis** et **psubst** permettent au contraire des substitutions en parallèle. Comparer les exemples suivants :

```
(%i11) sublis([x=b, b=c], a*x+c);

(%o11)                                   c + a b

(%i12) subst([x=b, b=c], a*x+c);

(%o12)                                   a c + c
```

Autre exemple :

```
(%i13) psubst ([a^2=b, b=a], sin(a^2) + sin(b));

(%o13)                             sin(b) + sin(a)

(%i14) subst ([a^2=b, b=a], sin(a^2) + sin(b));

(%o14)                                 2 sin(a)
```

L'option *opsubst*, si elle est fixée à FALSE, permet d'empêcher la substitution à l'intérieur d'un opérateur. Comparer les deux exemples qui suivent :

```
(%i7) (opsubst: false, subst (x^2, r, r+r[0])) ;
                                       2
(%o7)                                 x  + r
                                            0

(%i8) (opsubst: true, subst (x^2, r, r+r[0])) ;
                                      2     2
(%o8)                                x  + (x )
                                           0
```

• La fonction *ratsubst*

La fonction **ratsubst** a la même syntaxe que la fonction **subst** :

```
ratsubst (a, b, c)
```

Ici b peut être une somme, un produit, une puissance, etc. La fonction ne se limite pas à une simple substitution syntactique mais tient compte aussi de la signification des expressions. Par exemple, comparer :

```
(%i17) ratsubst (a, x + y, x + y + z) ;

(%o17)                                z + a

(%i18) subst (a, x + y, x + y + z) ;

(%o18)                              z + y + x
```

Voici quelques exemples d'utilisation :

```
(%i20) ratsubst (a, x*y^2, x*y^3 + x^2*y^4);

                                                2
(%o20)                              a y + a

(%i23) expr: cos(x)^4 + cos(x)^3 + cos(x)^2 + cos(x) + 1;

                      4         3         2
(%o23)             cos (x) + cos (x) + cos (x) + cos(x) + 1

(%i24) ratsubst (1 - sin(x)^2, cos(x)^2, expr);

                    4           2                       2
(%o24)          sin (x) - 3 sin (x) + cos(x) (2 - sin (x)) + 3
```

Lorsque l'option *radsubstflag* est fixée à TRUE, la fonction **ratsubst** fait des substitutions pour les radicaux même s'ils ne sont pas apparents. Par exemple :

```
(%i25) (radsubstflag:true, ratsubst(u,sqrt(x),x));

                                        2
(%o25)                                 u
```

3.3 Sous-expressions

• La fonction *part*

La fonction **part** renvoie des parties d'une expression. Celles-ci sont repérées par leur position dans la représentation interne sous forme de listes imbriquées. La syntaxe est :

```
part (expr, n_1, ..., n_k)
```

Elle extrait le n_1-ième terme, dont elle extrait ensuite le n_2-ième terme, dont elle prend le n_3-ième terme etc. La variable *piece* contient la dernière sous-expression calculée par la fonction **part**.

```
(%i72) expr: x^2-2*x*y^3+(z/a^2);
```

```
                                  z          3    2
(%o72)                            -- - 2 x y  + x
                                   2
                                  a

(%i73) part(expr,2,1);

                                        3
(%o73)                            2 x y

(%i74) part(expr,2);

                                         3
(%o74)                           - 2 x y

(%i75) part(expr,2,1,2);

(%o75)                               x

(%i76) part(expr,2,1,3);

                                      3
(%o76)                               y

(%i77) part(expr,2,1,3,2);

(%o77)                                3

(%i79) part(expr,[1,3]);

                                     z     2
(%o79)                               -- + x
                                      2
                                     a
```

L'indice passé en argument peut parfois être nul comme dans l'exemple suivant qui renvoie l'opérateur + :

```
(%i83) part (x+y, 0);

(%o83)                                +
```

• La fonction *op*

La fonction **op** renvoie l'opérateur principal d'une expression. Elle est équivalente à `part(expr,0)`.

• La fonction *dpart*

La fonction **dpart** renvoie les mêmes sous-expressions que **part** mais entourées d'une boîte. La fonction **lpart** ajoute une étiquette (*l* pour *label*).

```
dpart (expr, n_1,   , n_k)
lpart (label, expr, n_1,   , n_k)
```

• La fonction *inpart*

La fonction **inpart** fonctionne comme la fonction **part** mais opère sur la représentation interne. Elle est donc plus rapide puisqu'il n'y a pas de formatage des expressions.

La fonction **substpart** a pour syntaxe :

```
substpart (x, expr, n_1,..., n_k)
```

Elle fait la substitution par x de la sous-expression qui est extraite selon le même principe que pour la fonction **part**. Par exemple :

```
(%i92) expr: 1/(1+sin(x)^2);

                                      1
(%o92)                          -----------
                                   2
                                sin (x) + 1

(%i93) substpart(3,expr,2,1,2);

                                      1
(%o93)                          -----------
                                   3
                                sin (x) + 1
```

• La fonction *pickapart*

La fonction **pickapart** permet d'attribuer des étiquettes aux sous-expressions d'une expression à une certaine profondeur n. La syntaxe est :

```
pickapart(expr, n)
```

Les étiquettes ont la forme %tm où m est un nombre entier. Par exemple :

```
(%i101) expr: (a*x+b)/(%e^y+sin(z));

                                    a x + b
(%o101)                         ------------
                                            y
                                sin(z) + %e

(%i102) pickapart(expr,1);

(%t102)                            a x + b



                                            y
(%t103)                         sin(z) + %e

                                    %t102
(%o103)                             -----
                                    %t103

(%i106) pickapart(expr,2);
```

```
(%t106)                                  a x

(%t107)                                sin(z)

                                          y
(%t108)                                  %e

                                   b + %t106
(%o108)                           -------------
                                  %t108 + %t107
```

• La fonction *isolate*

La fonction **isolate** s'applique à une expression et renvoie cette expression en remplaçant les parties qui ne contiennent pas une certaine variable x par des étiquettes. L'intérêt est d'éviter d'appliquer des opérations à ces parties-là pendant un calcul pour ne travailler qu'avec celles qui contiennent la variable x. En fin de calcul, on peut remplacer les étiquettes par la sous-expression qu'elles représentent. La syntaxe est :

```
isolate(expr, x)
```

Par exemple :

```
(%i110) expr: (a*x+b)/(%e^y+x-sin(z^2));

                                      a x + b
(%o110)                        -------------------
                                      2      y
                               - sin(z ) + %e   + x

(%i111) isolate(expr,x);

                                    y        2
(%t111)                            %e  - sin(z )

                                     a x + b
(%o111)                              ---------
                                     x + %t111
```

• La fonction *disolate*

La fonction **disolate** est similaire mais permet d'isoler plusieurs variables à la fois. La syntaxe est :

```
disolate (expr, x_1,..., x_n)
```

• La fonction *partition*

La fonction **partition** sépare dans un produit, une somme ou une liste, les éléments qui contiennent une certaine variable et les autres. Par exemple :

```
(%i114) partition((x+a)*(y+b),x);

(%o114)                          [y + b, x + a]

(%i115) partition((x+a)+(y+b),x);

(%o115)                          [y + b + a, x]

(%i116) partition([a,x+b,f(x)+c,d],x);

(%o116)                  [[a, d], [x + b, f(x) + c]]
```

3.4 Simplification

• La fonction *expand*

La fonction la plus utilisée pour développer et simplifier une expressions algébrique est la fonction **expand** dont la syntaxe générale est :

```
expand (expr)
expand (expr, p, n)
```

Dans la deuxième forme, les arguments p et n spécifient respectivement les exposants positifs et négatifs maximaux qui doivent être développés. Ces valeurs sont contrôlées au niveau global par les options *maxposex* et *maxnegex*.

Par exemple :

```
(%i1) expand( (a+x)^3/(x+1)^2 );

          3                 2             2              3
         x             3 a x           3 a  x           a
(%o1)  ------------ + ------------ + ------------ + ------------
        2              2              2              2
       x  + 2 x + 1   x  + 2 x + 1   x  + 2 x + 1   x  + 2 x + 1

(%i2) expand( (a+x)^3/(x+1)^2 ,2,2);

                           3
                     (x + a)
(%o2)              ------------
                    2
                   x  + 2 x + 1

(%i3) expand( (a+x)^3/(x+1)^2 ,2,1);

                           3
                     (x + a)
(%o3)                --------
                           2
                     (x + 1)
```

Pour des polynômes, voir aussi la fonction **ratexpand** à la section 6.6.

• La fonction *expandwrt*

La fonction **expandwrt** permet de développer une expression par rapport à certaines variables $x_1, \ldots, x_n$. La syntaxe générale est :

```
expandwrt(expr, x_1,..., x_n)
```

Par exemple :

```
(%i4) expandwrt( (A+x)*(B+y), x );

(%o4)                          A (B + y) + x (B + y)
```

La fonction **expandwrt_factored** est similaire mais ne développe que sur les facteurs qui contiennent les variables indiquées.

• La fonction *distrib*

La fonction **distrib** développe une expression par distributivité mais diffère de la fonction **expand** en ce qu'elle n'opère qu'au premier niveau, sans récursivité. Par exemple :

```
(%i61) distrib ((a+b) * (c+d));

(%o61)                          b d + a d + b c + a c
```

• La fonction *multthru*

La fonction **multthru** développe des expressions qui sont des produits de facteurs dans lesquels l'un des facteurs est une somme. Chaque terme de la somme est multiplié par les autres facteurs. La syntaxe générale prend deux formes :

```
multthru (expr)
multthru (expr_1, expr_2)
```

Dans la deuxième forme, la deuxième expression doit être une somme et chacun de ses termes est multiplié par la première expression. Par exemple :

```
(%i14) multthru( (a+b),(x^2+x+1) );

                    2
(%o14)  (b + a) x  + (b + a) x + b + a

(%i15) multthru( (a+b)*(c+d)*(x^2+x+1) );

                   2
(%o15)     (b + a) (d + c) x  + (b + a) (d + c) x + (b + a) (d + c)
```

• La fonction *scsimp*

La fonction **scsimp** (abréviation de *Sequential Comparative Simplification*) permet de simplifier une expression compte-tenu de certaines relations. Par exemple :

```
(%i17) exp:-k^2*l^2*m^2*n^2-k^2*l^2*n^2+k^2*m^2*n^2+k^2*n^2
```

```
                      2  2  2  2    2  2  2    2  2  2    2  2
(%o17)           - k  l  m  n  + k  m  n  - k  l  n  + k  n
```

```
(%i18) eq1:l^2+k^2 = 1
```

```
                                    2    2
(%o18)                             l  + k  = 1
```

```
(%i19) eq2:n^2-m^2 = 1
```

```
                                    2    2
(%o19)                             n  - m  = 1
```

```
(%i20) scsimp(exp,eq1,eq2)
```

```
                                      4  4
(%o20)                               k  n
```

• La fonction *combine*

La fonction **combine** simplifie une expression en combinant entre eux tous les termes ayant le même dénominateur. Par exemple :

```
(%i28) combine( a/d+b/c+c/d);
```

```
                    c + a   b
(%o28)              ----- + -
                      d     c
```

• La fonction *xthru*

La fonction **xthru** simplifie les facteurs communs entre numérateur et dénominateur comme la fonction **ratsimp** mais seulement si les facteurs sont explicites. Par exemple :

```
(%i56) xthru(1/x+2/x^2+3/x^3);
```

```
                                   2
                                  x  + 2 x + 3
(%o56)                            ------------
                                        3
                                       x
```

```
(%i60) xthru(1/c+1/d);
```

```
                                     d + c
(%o60)                               -----
                                      c d
```

• La fonction *exponentialize*

La fonction **exponentialize** convertit les fonctions trigonométriques au moyen des formules d'Euler. Par exemple :

```
(%i12) exponentialize(sin(x)+cos(x));

                    %i x     - %i x          %i x     - %i x
                   %e     + %e          %i (%e     - %e      )
(%o12)             ----------------- - ----------------------
                           2                      2
```

• La fonction *radcan*

La fonction **radcan** simplifie des expressions contenant des logarithmes, des exponentielles et des radicaux en les mettant sous une forme canonique. Par exemple :

```
(%i54) radcan(sqrt(6)/sqrt(2));

(%o54)                                sqrt(3)

(%i55) radcan((%e^x-1)/(1+%e^(x/2)));

                                        x/2
(%o55)                                %e    - 1
```

3.5 Anatomie d'une expression

Maxima distingue deux sortes d'opérateurs :
- les verbes sont des opérateurs qui peuvent être exécutés. Les fonctions sont par défaut considérées comme des verbes.
- les noms (*nouns*) jouent le rôle de prédicats. Ce sont des opérateurs qui apparaissent comme des symboles dans une expression mais ne sont pas exécutés.

Les fonctions **verbify** et **nounify** permettent de modifier le type d'un opérateur.

• La fonction *args*

La fonction **args** renvoie une liste des arguments de l'expression qu'elle prend en argument tandis que la fonction **nterms** renvoie le nombre des arguments :

```
(%i28) expr: x^4+x^3-x^2-x+1;

                              4    3    2
(%o28)                       x  + x  - x  - x + 1

(%i29) nterms(expr);

(%o29)                                5

(%i30) args(expr);

                              4   3     2
(%o30)                       [x , x , - x , - x, 1]
```

• La fonction *atom*

La fonction **atom** renvoie TRUE si l'expression est atomique et FALSE sinon.

• La fonction *symbolp*

La fonction **symbolp** renvoie TRUE si l'expression est un symbole et FALSE sinon.

• La fonction *listofvars*

La fonction **listofvars** renvoie une liste des variables contenues dans une expression :

```
(%i32) listofvars(a*x^2+b*y+c*%e^z);

(%o32)                          [a, x, b, y, c, z]
```

Les options *listconstvars* et *listdummyvars* permettent de modifier le comportement de cette fonction ;

```
(%i35) (listconstvars:true, listofvars(a*x^2+b*y+c*%e^z));

(%o35)                       [a, x, b, y, c, %e, z]
```

• La fonction *reveal*

La fonction **reveal** permet d'analyser la nature des termes qui composent une expression. La syntaxe générale est :

```
reveal (expr, depth)
```

L'argument *depth* indique le niveau de récursivité de la fonction. Par exemple :

```
(%i43) expr: (x^2+x*y)/(%e^x-(a/b)^3);

                                                2
                                           x y + x
(%o43)                                     --------
                                                3
                                             x   a
                                           %e  - --
                                                3
                                               b

(%i46) reveal(expr,1);

(%o46)                                     Quotient

(%i47) reveal(expr,2);

                                            Sum(2)
(%o47)                                      ------
                                            Sum(2)

(%i48) reveal(expr,3);

                                       Product(2) + Expt
(%o48)                                 -----------------
                                        Expt + Negterm
```

```
(%i49) reveal(expr,4);

                                        2
                                   x y + x
(%o49)                           --------------
                                     x
                                 %e  - Quotient

(%i50) reveal(expr,5);

                                        2
                                   x y + x
(%o50)                           ----------
                                   x   Expt
                                 %e  - ----
                                       Expt
```

• La fonction *freeof*

La fonction **freeof** permet de savoir si certaines variables sont présentes ou pas dans une expression. La syntaxe est :

```
freeof (x_1,   , x_n, expr)
```

La fonction renvoie TRUE si aucune sous-expression ne contient x_i ou si x_i intervient seulement comme une variable muette ou encore si elle n'est ni la forme nominale ou verbale d'un opérateur. Par exemple :

```
(%i96) expr: a*x^2+sin(z);

                                          2
(%o96)                          sin(z) + a x

(%i97) freeof(a,expr);

(%o97)                              false

(%i98) freeof(sin,expr);

(%o98)                              false

(%i99) freeof(cos,expr);

(%o99)                              true
```

• La fonction *lfreeof*

La fonction **lfreeof** prend en argument une liste et applique la fonction **freeof** à chaque membre de la liste. Par exemple :

```
(%i100) lfreeof([cos,y,b+x],expr);

(%o100)                             true
```

• La fonction *ordergreat*

Les fonctions **ordergreat** et **orderless** permettent de modifier l'ordre canonique des éléments dans les expressions. La fonction **unorder** rétablit l'ordre par défaut.

• La fonction *optimize*

La fonction **optimize** décompose une expression en éléments à évaluer au moyen de la fonction **block**. Elle le fait de manière à obtenir une décomposition efficace qui évite de recalculer ou de réévaluer plusieurs fois des éléments communs.

```
(%i17) expr: %e^{x^2+y^2}+1/(x^2+y^2)-sqrt(x^2+y^2+1);

                        2    2
                      {y  + x }          2    2            1
(%o17)              %e           - sqrt(y  + x  + 1) + -------
                                                      2    2
                                                     y  + x

(%i18) optimize(%);

                                    2        2
(%o18) block([%1, %2, %3], %1 : x , %2 : y , %3 : %2 + %1,
                                      {%3}                        1
                                   %e     - sqrt(%2 + %1 + 1) + --)
                                                                 %3
```

4 Nombres complexes

Les nombres complexes sont notés, sous forme algébrique, au moyen du symbole %i. Par exemple :

```
(%i13) z: 2+3*%i;

(%o13)                              3 %i + 2
```

Le conjugué est obtenu au moyen de la fonction **conjugate** comme ceci :

```
(%i25) conjugate(z);

(%o25)                              2 - 3 %i
```

Les fonctions **realpart** et **imagpart** renvoient respectivement la partie réelle et la partie imaginaire d'un nombre complexe. Par exemple :

```
(%i14) realpart(z);

(%o14)                                2

(%i15) imagpart(z);

(%o15)                                3
```

La fonction **polarform** transforme un nombre complexe de sa forme algébrique à sa forme exponentielle. Par exemple :

```
(%i16) polarform(z);

                                          %i atan(3/2)
(%o16)                          sqrt(13) %e
```

La fonction **rectform** effectue la transformation inverse. Par exemple :

```
(%i22) w: 2*%e^(%i*%pi/4);

                                    %i          1
(%o22)                         2 (------- + -------)
                                  sqrt(2)    sqrt(2)

(%i23) rectform(w);

(%o23)                         sqrt(2) %i + sqrt(2)
```

Les fonctions **abs** et **cabs** calculent le module d'un nombre complexe :

```
(%i27) cabs(z);

(%o27)                                 sqrt(13)
```

La fonction **carg** renvoie l'argument d'un complexe dans l'intervalle $]-\pi,\pi]$:

```
(%i28) carg(z);

                                          3
(%o28)                                atan(-)
                                          2

(%i29) carg(w);

                                        %pi
(%o29)                                  ---
                                         4
```

• La fonction *demoivre*

La fonction **demoivre** convertit un nombre complexe de la notation exponentielle à la notation trigonométrique. Par exemple :

```
(%i10) demoivre (exp (2*%i * x));

(%o10)                         %i sin(2 x) + cos(2 x)
```

5 Matrices

Dans Maxima, une matrice est représentée par une liste de sous-listes, chaque sous-liste représentant un rang de la matrice.

5.1 Construction de matrices

La fonction **matrix** permet de créer une matrice : elle prend en argument les rangs de la matrice. Par exemple :

```
(%i2) M: matrix([2,3,4],[-1,0,2], [5,0,2]);

                                  [  2   3  4 ]
                                  [           ]
(%o2)                             [ - 1  0  2 ]
                                  [           ]
                                  [  5   0  2 ]
```

• La fonction *entermatrix*

La fonction **entermatrix** permet d'entrer interactivement les éléments de la matrice. Elle prend en arguments le nombre de lignes et le nombre de colonnes et tient compte de la forme éventuelle de la matrice (diagonale, symétrique, etc). Par exemple :

```
(%i7) N: entermatrix(2,2);


Is the matrix  1. Diagonal  2. Symmetric  3. Antisymmetric  4. General
Answer 1, 2, 3 or 4 :
4;
Row 1 Column 1:
1;
Row 1 Column 2:
3;
Row 2 Column 1:
-3;
Row 2 Column 2:
m;

Matrix entered.
                                  [  1   3 ]
(%o7)                             [        ]
                                  [ - 3  m ]
```

• La fonction *genmatrix*

La fonction **genmatrix** permet de construire une matrice à partir d'un tableau de dimension 2. Par exemple, les matrices de Hilbert peuvent être obtenues comme ceci :

```
(%i12) h[i,j] := 1 / (i + j - 1);

                                       1
(%o12)                  h      := ---------
                         i, j     i + j - 1

(%i13) H: genmatrix(h,3,3);
```

```
                                    [     1   1 ]
                                    [ 1   -   - ]
                                    [     2   3 ]
                                    [           ]
                                    [ 1   1   1 ]
(%o13)                              [ -   -   - ]
                                    [ 2   3   4 ]
                                    [           ]
                                    [ 1   1   1 ]
                                    [ -   -   - ]
                                    [ 3   4   5 ]
```

La syntaxe complète de la fonction est :

```
genmatrix (a, i_2, j_2, i_1, j_1)
```

Les indices i_1 et j_1 représentent le terme supérieur gauche et les indices i_2 et j_2 le coin inférieur droit. Par exemple :

```
(%i14) H: genmatrix(h,3,4,2,2);

                                    [ 1   1   1 ]
                                    [ -   -   - ]
                                    [ 3   4   5 ]
(%o14)                              [           ]
                                    [ 1   1   1 ]
                                    [ -   -   - ]
                                    [ 4   5   6 ]
```

Les fonctions **zeromatrix** et **ident** fabriquent respectivement des matrices nulles et des matrices unité :

```
(%i17) zeromatrix(3,2);

                                     [ 0   0 ]
                                     [       ]
(%o17)                               [ 0   0 ]
                                     [       ]
                                     [ 0   0 ]

(%i18) ident(3);

                                    [ 1   0   0 ]
                                    [           ]
(%o18)                              [ 0   1   0 ]
                                    [           ]
                                    [ 0   0   1 ]
```

• La fonction *diagmatrix*

La fonction **diagmatrix** crée une matrice diagonale en reproduisant n fois un élément x donné. L'élément x peut être une expression ou une autre matrice. Par exemple :

```
(%i26) D: diagmatrix(2,P);

                                    [ P  0 ]
(%o26)                              [      ]
                                    [ 0  P ]
```

Les fonctions **addrow** et **addcol** produisent une nouvelle matrice en ajoutant respectivement des rangs ou des colonnes à une matrice. Par exemple :

```
(%i20) addcol(N,[m^2,1]);

                                [          2 ]
(%o20)                          [  1   3   m ]
                                [            ]
                                [ - 3  m   1 ]

(%i22) addrow(N,[m^2,1]);

                                  [  1   3 ]
                                  [        ]
(%o22)                            [ - 3  m ]
                                  [        ]
                                  [  2     ]
                                  [ m    1 ]
```

• La fonction *submatrix*

La fonction **submatrix** permet d'extraire des sous-matrices d'une matrice. La syntaxe complète est :

```
submatrix (i_1, ..., i_m, M, j_1, ..., j_n)
```

On indique les indices i des rangs et les indices j des colonnes qui doivent être supprimés. On peut omettre les indices i ou j si aucun rang ou colonne n'est supprimé. Par exemple :

```
(%i29) submatrix(1,2,M,3);

(%o29)                              [ 5  0 ]
```

• La fonction *copymatrix*

La fonction **copymatrix** renvoie une copie d'une matrice.

```
(%i28) M2: copymatrix (M);

                                [  2   3  4 ]
                                [           ]
(%o28)                          [ - 1  0  2 ]
                                [           ]
                                [  5   0  2 ]
```

• La fonction *cauchy_matrix*

La fonction **cauchy_matrix** construit une matrice de Cauchy de terme général $\frac{1}{x_i + y_j}$. La syntaxe générale prend deux formes :

```
cauchy_matrix ([x_1, x_2, ..., x_m], [y_1, y_2, ..., y_n])
cauchy_matrix ([x_1, x_2, ..., x_n])
```

Lorsqu'on omet le deuxième argument, la matrice a pour terme général $\frac{1}{x_i + x_j}$. Par exemple :

```
(%i62) cauchy_matrix([m,n],[p,q]);
```

$$(\%o62)\quad \begin{bmatrix} \frac{1}{p+m} & \frac{1}{q+m} \\ \frac{1}{p+n} & \frac{1}{q+n} \end{bmatrix}$$

```
(%i63) cauchy_matrix([m,n]);
```

$$(\%o63)\quad \begin{bmatrix} \frac{1}{2\,m} & \frac{1}{n+m} \\ \frac{1}{n+m} & \frac{1}{2\,n} \end{bmatrix}$$

• La fonction *columnvector*

La fonction **columnvector** (ou, de manière synonyme, **covect**) permet de fabriquer une matrice colonne à partir d'une liste :

```
(%i27) covect([a,z,e]);
```

$$(\%o27)\quad \begin{bmatrix} a \\ z \\ e \end{bmatrix}$$

Les fonctions **row** et **col** renvoient respectivement le rang ou la colonne d'indice i dans une matrice. La syntaxe est :

```
col (M, i)
row (M, i)
```

Par exemple :

```
(%i24) col(M,2);
```

```
                                        [ 3 ]
                                        [   ]
(%o24)                                  [ 0 ]
                                        [   ]
                                        [ 0 ]

(%i26) row(M,2);

(%o26)                          [ - 1  0  2 ]
```

• La fonction *ematrix*

La fonction **ematrix** construit une matrice nulle à l'exception d'un élément. La syntaxe est :

```
ematrix (m, n, x, i, j)
```

L'élément d'indice (i, j) est égal à l'argument spécifié x. Par exemple :

```
(%i1) ematrix(2,2,m^2,1,2);

                    [     2 ]
(%o1)               [ 0   m ]
                    [       ]
                    [ 0   0 ]
```

• La fonction *matrixp*

La fonction **matrixp** teste si un objet est une matrice. Elle renvoie TRUE si l'argument est une matrice, FALSE sinon.

```
(%i4) matrixp(A);

(%o4)                                   true
(%i5) matrixp(%pi);
(%o5)                                   false
```

5.2 Opérations sur les matrices

Le produit matriciel est désigné par un point. Par exemple :

```
(%i31) A: matrix([3,2,1],[1,-2,3]);

                                    [ 3   2   1 ]
(%o31)                              [           ]
                                    [ 1  - 2  3 ]

(%i32) B: matrix([1,1],[2,-2],[0,-3]);

                                     [ 1   1  ]
                                     [        ]
(%o32)                               [ 2  - 2 ]
                                     [        ]
                                     [ 0  - 3 ]
```

```
(%i33) P: A.B;

                                    [  7   - 4 ]
(%o33)                              [          ]
                                    [ - 3  - 4 ]
```

• La fonction *invert*

La fonction **invert** calcule l'inverse d'une matrice carrée. Par exemple :

```
(%i36) invert(P);

                                  [  1      1  ]
                                  [  --   - -- ]
                                  [  10     10 ]
(%o36)                            [            ]
                                  [   3     7  ]
                                  [ - --  - -- ]
                                  [   40    40 ]
```

L'option *detout* permet d'avoir le déterminant à l'extérieur de la matrice inverse :

```
(%i37) invert(P), detout;

                                    [ - 4  4 ]
                                    [        ]
                                    [  3   7 ]
(%o37)                            - ----------
                                        40
```

• La fonction *transpose*

La fonction **transpose** renvoie la matrice transposée :

```
(%i38) transpose(B);

                                  [ 1   2    0  ]
(%o38)                            [             ]
                                  [ 1  - 2  - 3 ]
```

Le déterminant est calculé au moyen de la fonction **determinant** :

```
(%i39) P;

                                    [  7   - 4 ]
(%o39)                              [          ]
                                    [ - 3  - 4 ]
(%i40) determinant(P);
(%o40)                                - 40
```

On peut aussi utiliser la fonction **newdet** qui effectue le calcul par la méthode de Johnson-Gentleman. Le résultat est renvoyé en format CRE. La fonction **permanent** utilise le même algorithme pour calculer le permanent d'une matrice : c'est la même chose que le déterminant sans les changements de signe. Le résultat est en format CRE.

```
(%i33) newdet(N);

(%o33)/R/                                m + 9

(%i34) permanent(N);

(%o34)/R/                                m - 9
```

• La fonction *mattrace*

La fonction **mattrace** renvoie la trace d'une matrice carrée, c'est-à-dire la somme des éléments de la diagonale principale. Par exemple :

```
mattrace(P);
```

• La fonction *rank*

La fonction **rank** calcule le rang d'une matrice :

```
(%i49) rank(B);

(%o49)                                    2
```

• La fonction *minor*

La fonction **minor** renvoie le mineur d'une matrice obtenu en supprimant une ligne et une colonne d'indices donnés. La syntaxe est :

```
minor(M, i, j)
```

où i et j désignent respectivement l'indice de la ligne et de la colonne à supprimer. Par exemple :

```
(%i52) minor(M,2,3);

                                       [ 2  3 ]
(%o52)                                 [      ]
                                       [ 5  0 ]
```

• La fonction *adjoint*

La fonction **adjoint** renvoie la transposée de la matrice des cofacteurs. Par exemple :

```
(%i55) M;

                                    [  2   3  4 ]
                                    [           ]
(%o55)                              [ - 1  0  2 ]
                                    [           ]
                                    [  5   0  2 ]

(%i56) adjoint(M);
```

```
                                [ 0   - 6    6  ]
                                [               ]
(%o56)                          [ 12  - 16  - 8 ]
                                [               ]
                                [ 0    15    3  ]
```

L'inverse d'une matrice est la matrice adjointe divisée par le déterminant. Par exemple :

```
(%i57) determinant(M);

(%o57)                                36

(%i58) invert(M),detout;

                                [ 0   - 6    6  ]
                                [               ]
                                [ 12  - 16  - 8 ]
                                [               ]
                                [ 0    15    3  ]
(%o58)                          -----------------
                                       36
```

• La fonction *setelmx*

La fonction **setelmx** permet de modifier l'élément (i, j) d'une matrice. La syntaxe est

```
setelmx (x, i, j, M)
```

L'élément m_{ij} est remplacé par l'argument x.

• La fonction *matrixmap*

La fonction **matrixmap** applique une fonction f à tous les éléments d'une matrice. Elle renvoie la matrice de terme général $f(m_{ij})$. Par exemple :

```
(%i6) matrixmap(g,A);

                [ g(3)   g(2)   g(1) ]
(%o6)           [                    ]
                [ g(1)  g(- 2)  g(3) ]
```

5.3 Opérations sur les vecteurs

• La fonction *innerproduct*

La fonction **innerproduct** (ou, de manière synonyme, **inprod**) calcule le produit scalaire de deux vecteurs.

```
(%i6) innerproduct([1,2,2],[2,-2,1]);

(%o6)                                 0
```

• La fonction *unitvector*

La fonction **unitvector** (ou, de manière synonyme, **uvect**) permet de normer un vecteur :

```
(%i5) unitvector([1,2,2]);

                                    1  2  2
(%o5)                              [-, -, -]
                                    3  3  3
```

• La fonction *gramschmidt*

La fonction **gramschmidt** applique le procédé d'orthogonalisation de Gram-Schmidt sur une matrice (ou sur une liste de listes). La syntaxe générale est :

```
gramschmidt (x)
gramschmidt (x, F)
```

Dans la deuxième forme, F désigne le produit scalaire à utiliser. Par défaut, c'est le produit scalaire euclidien ordinaire. Il faut charger le package **eigen** pour pouvoir utiliser cette fonction :

```
(%i2) load(eigen)$
(%i3) gramschmidt([[1,1,0],[1,0,1],[0,1,1]]);

                                 1    1         2  2  2
(%o3)              [[1, 1, 0], [-, - -, 1], [- -, -, -]]
                                 2    2         3  3  3
```

• La fonction *vectorsimp*

La fonction **vectorsimp** effectue des simplifications sur un vecteur, compte-tenu de la valeur des options booléennes suivantes : *expandall*, *expanddot*, *expanddotplus*, *expandcross*, *expandcrossplus*, *expandcrosscross*, *expandgrad*, *expandgradplus*, *expandgradprod*, *expanddiv*, *expanddivplus*, *expanddivprod*, *expandcurl*, *expandcurlplus*, *expandcurlcurl*, *expandlaplacian*, *expandlaplacianplus*, *expandlaplacianprod*.

• La fonction *list_matrix_entries*

La fonction **list_matrix_entries** renvoie une liste contenant les éléments d'une matrice énumérés en lignes. Par exemple :

```
(%i2) A: matrix([3,2,1],[1,-2,3]);

                 [ 3   2   1 ]
(%o2)            [           ]
                 [ 1  - 2  3 ]

(%i3) list_matrix_entries(A);

(%o3)                       [3, 2, 1, 1, - 2, 3]
```

5.4 Valeurs propres et vecteurs propres

• La fonction *eigenvalues*

La fonction **eigenvalues** (ou, de manière synonyme, **eivals**) renvoie les valeurs propres d'une matrice sous la forme d'une liste faite de deux sous-listes :

- la première sous-liste contient les valeurs propres elles-mêmes ;
- la deuxième sous-liste contient leurs multiplicités.

• La fonction *eigenvectors*

La fonction **eigenvectors** (ou, de manière synonyme **eivects**) calcule les vecteurs propres d'une matrice carrée. Elle renvoie une liste de deux éléments :

- le premier élément est une liste comportant les valeurs propres et leurs multiplicités comme avec la fonction **eigenvalues** ;
- le second élément est une liste de sous-listes : chaque sous-liste contient les vecteurs propres associés à la valeur propre correspondante.

• La fonction *uniteigenvectors*

La fonction **uniteigenvectors** (ou, de manière synonyme, **ueivects**) est similaire à la fonction **eigenvectors** mais renvoie des vecteurs normés.

Par exemple :

```
(%i17) R: matrix([1,2],[2,3]);

                                    [ 1  2 ]
(%o17)                              [      ]
                                    [ 2  3 ]

(%i19) eigenvalues(R);

(%o19)              [[2 - sqrt(5), sqrt(5) + 2], [1, 1]]

(%i20) eigenvectors(R);

(%o20) [[[2 - sqrt(5), sqrt(5) + 2], [1, 1]],
                         sqrt(5) - 1          sqrt(5) + 1
               [[[1, - -----------]], [[1, -----------]]]]
                              2                    2
```

• La fonction *charpoly*

La fonction **charpoly** calcule le polynôme caractéristique d'une matrice carrée. Le second argument désigne le nom de la variable à utiliser. Par exemple :

```
(%i2) p: charpoly(R,lambda);

(%o2)                    (1 - lambda) (3 - lambda) - 4

(%i3) expand(p);

                                   2
(%o3)                       lambda  - 4 lambda - 1
```

On peut aussi utiliser la fonction **ncharpoly** qui calcule ce polynôme par une autre méthode.

• La fonction *eigens_by_jacobi*

La fonction **eigens_by_jacobi** fait partie du package *linearalgebra*. Elle calcule numériquement les valeurs propres et les vecteurs propres d'une matrice symétrique par la méthode des rotations de Jacobi. Par exemple :

```
(%i49) A: matrix([ 4, 5],[5, 4]);

                                    [ 4  5 ]
(%o49)                              [      ]
                                    [ 5  4 ]

(%i50) eigens_by_jacobi(A);

                          [  .7071067811865476   .7071067811865475 ]
(%o50)    [[- 1.0, 9.0], [                                        ]]
                          [ - .7071067811865475  .7071067811865476 ]
```

5.5 Transformations matricielles

• La fonction *coefmatrix*

La fonction **coefmatrix** extrait la matrice d'un système d'équations linéaires. La syntaxe générale est :

```
coefmatrix ([eqn_1, ..., eqn_m], [x_1, ..., x_n])
```

Par exemple :

```
(%i7) coefmatrix([2*x-(m-1)*y+5*b = 0, m^2*y+(m+1)*x = 3], [x,y]);

                                  [   2     1 - m ]
(%o7)                             [                ]
                                  [           2    ]
                                  [ m + 1    m     ]
```

• La fonction *augcoefmatrix*

La fonction **augcoefmatrix** fonctionne comme **coefmatrix** mais ajoute une colonne représentant le membre de droite du système d'équations :

```
(%i8) augcoefmatrix([2*x-(m-1)*y+5*b = 0, m^2*y+(m+1)*x = 3], [x,y]);

                                [   2     1 - m   5 b ]
(%o8)                           [                     ]
                                [           2         ]
                                [ m + 1    m      - 3 ]
```

• La fonction *similaritytransform*

La fonction **similaritytransform** (ou, de manière synonyme, **simtran**) calcule une matrice semblable à une matrice M donnée. Elle renvoie une liste qui est le résultat de la fonction **uniteigenvectors**. Si M est diagonalisable (autrement dit, si l'option *nondiagonalizable* est égale à FALSE), elle calcule aussi les matrices P et Q telles que PMQ soit la matrice diagonale. Si l'option *hermitianmatrix* est égale à TRUE, P est la conjuguée de la transposée de Q, autrement P est l'inverse de Q. Les colonnes de Q sont les vecteurs propres normés.

Il faut charger le package **eigen** pour pouvoir utiliser cette fonction :

```
(%i14) load (eigen)$
(%i15) R: matrix([1,2],[2,1]);

                                    [ 1  2 ]
(%o15)                              [      ]
                                    [ 2  1 ]

(%i16) similaritytransform(R);

                                  1         1           1          1
(%o16) [[[3, - 1], [1, 1]], [[[-------, -------]], [[-------, - -------]]]]
                                sqrt(2)   sqrt(2)     sqrt(2)    sqrt(2)
```

Les fonctions **echelon** et **triangularize** renvoient une matrice triangulaire obtenue à partir d'une matrice donnée par élimination par la méthode de Gauss. Dans le cas de la fonction **echelon**, les coefficients diagonaux sont normalisés à 1.

```
(%i28) S: matrix([1,2,2],[2,1,2],[2,2,1]);

                                  [ 1  2  2 ]
                                  [         ]
(%o28)                            [ 2  1  2 ]
                                  [         ]
                                  [ 2  2  1 ]

(%i29) echelon(S);

                                  [ 1  2  2 ]
                                  [         ]
                                  [       2 ]
(%o29)                            [ 0  1  - ]
                                  [       3 ]
                                  [         ]
                                  [ 0  0  1 ]

(%i30) triangularize(S);

                                [ 1   2    2  ]
                                [             ]
(%o30)                          [ 0  - 3  - 2 ]
                                [             ]
                                [ 0   0    5  ]
```

Les fonctions **lu_factor** et **cholesky** permettent aussi d'obtenir des décompositions triangulaires. Elles sont définies dans le package **linearalgebra** (voir à la section 5.7).

5.6 Options de calcul matriciel

Un certain nombre d'options permettent de paramétrer la manière dont les calculs matriciels sont effectués. Voir en particulier :

detout	*dotdistrib*
doallmxops	*dotexptsimp*
domxexpt	*dotident*
domxmxops	*dotscrules*
domxnctimes	*lmxchar*
dontfactor	*matrix_element_add*
doscmxops	*matrix_element_mult*
doscmxplus	*matrix_element_transpose*
dot0nscsimp	*ratmx*
dot0simp	*rmxchar*
dot1simp	*scalarmatrixp*
dotassoc	*sparse*
dotconstrules	*vect_cross*

5.7 Le package *linearalgebra*

Le package est chargé au moyen de l'instruction :

```
load(linearalgebra);
```

Lorsqu'une de ses fonctions est invoquée, il est automatiquement chargé donc on peut aussi l'utiliser directement sans exécuter la commande **load**.

```
(%i7) M: matrix([1,2,3],[4,5,6],[7,8,9]);

                                    [ 1  2  3 ]
                                    [         ]
(%o7)                               [ 4  5  6 ]
                                    [         ]
                                    [ 7  8  9 ]
```

Dans les exemples qui suivent, on utilise la matrice M suivante :

```
(%i7) M: matrix([1,2,3],[4,5,6],[7,8,9]);

                                    [ 1  2  3 ]
                                    [         ]
(%o7)                               [ 4  5  6 ]
                                    [         ]
                                    [ 7  8  9 ]
```

5.7.1 Opérations matricielles

Les fonctions **rowswap** et **columnswap** permettent respectivement de commuter des lignes ou des colonnes d'une matrice. Par exemple :

```
(%i8) rowswap(M,2,3);
```

```
                                    [ 1  2  3 ]
                                    [         ]
(%o8)                               [ 7  8  9 ]
                                    [         ]
                                    [ 4  5  6 ]
```

```
(%i9) columnswap(M,1,3);

                                    [ 3  2  1 ]
                                    [         ]
(%o9)                               [ 6  5  4 ]
                                    [         ]
                                    [ 9  8  7 ]
```

Les fonctions **columnop** et **rowop** renvoient le résultat de la combinaison linéaire d'une ligne ou colonne avec une autre. Par exemple, la commande *columnop(M,i,j,t)* remplace la colonne C_i par $C_i - t * C_j$. Voici deux exemples :

```
(%i11) rowop(M,2,3,t);

                          [     1         2         3     ]
                          [                               ]
(%o11)                    [ 4 - 7 t   5 - 8 t   6 - 9 t ]
                          [                               ]
                          [     7         8         9     ]
```

```
(%i10) columnop(M,1,3,x);

                                [ 1 - 3 x  2  3 ]
                                [               ]
(%o10)                          [ 4 - 6 x  5  6 ]
                                [               ]
                                [ 7 - 9 x  8  9 ]
```

```
(%i10) columnop(M,2,3,2/3);

                                   [ 1  0  3 ]
                                   [         ]
(%o10)                             [ 4  1  6 ]
                                   [         ]
                                   [ 7  2  9 ]
```

• La fonction *dotproduct*

La fonction **dotproduct** effectue le produit scalaire de deux vecteurs réels ou le produit hermitien dans le cas de vecteurs complexes :

```
(%i3) u: matrix([1],[2],[3]);

                                      [ 1 ]
                                      [   ]
(%o3)                                 [ 2 ]
                                      [   ]
                                      [ 3 ]
```

```
(%i4) v: matrix([-1],[0],[2]);

                                     [ - 1 ]
                                     [     ]
(%o4)                                [  0  ]
                                     [     ]
                                     [  2  ]

(%i5) dotproduct(u,v);

(%o5)                                  5
```

Dans le cas complexe :

```
(%i6) uc: matrix([1+%i],[2],[3*%i]);

                                   [ %i + 1 ]
                                   [        ]
(%o6)                              [   2    ]
                                   [        ]
                                   [  3 %i  ]

(%i7) vc: matrix([-%i],[0],[2]);

                                    [ - %i ]
                                    [      ]
(%o7)                               [  0   ]
                                    [      ]
                                    [  2   ]

(%i8) dotproduct(uc,vc);

(%o8)                        - (1 - %i) %i - 6 %i

(%i9) ratsimp(%);

(%o9)                             - 7 %i - 1
```

• La fonction *kronecker_product*

La fonction **kronecker_product** effectue le produit de Kronecker de deux matrices :

```
(%i10) A: matrix([a,b],[c,d]);

                                    [ a  b ]
(%o10)                              [      ]
                                    [ c  d ]

(%i11) B: matrix([1,2],[3,4]);

                                    [ 1  2 ]
(%o11)                              [      ]
                                    [ 3  4 ]
```

```
(%i12) kronecker_product(A,B);

                            [  a   2 a   b   2 b ]
                            [                    ]
                            [ 3 a  4 a  3 b  4 b ]
(%o12)                      [                    ]
                            [  c   2 c   d   2 d ]
                            [                    ]
                            [ 3 c  4 c  3 d  4 d ]
```

• La fonction *ctranspose*

La fonction **ctranspose** renvoie la conjuguée de la transposée d'une matrice complexe :

```
(%i1) C: matrix([2+%i,2-%i],[%i,4]);

                               [ %i + 2  2 - %i ]
(%o1)                          [                ]
                               [   %i      4    ]

(%i2) ctranspose(C);

                                [ 2 - %i  - %i ]
(%o2)                           [              ]
                                [ %i + 2   4   ]
```

• La fonction *addmatrices*

La fonction **addmatrices** combine deux matrices en appliquant une fonction à chacun de leurs termes. Par exemple :

```
(%i44) m1 : matrix([1,2],[13,14]);

                                  [ 1   2  ]
(%o44)                            [        ]
                                  [ 13  14 ]

(%i45) m2 : matrix([7,8],[9,10]);

                                   [ 7  8  ]
(%o45)                             [       ]
                                   [ 9  10 ]

(%i46) addmatrices('max,m1,m2);

                                  [ 7   8  ]
(%o46)                            [        ]
                                  [ 13  14 ]

(%i47) addmatrices('max,m1,m2,m2-m1);

                                  [ 7   8  ]
(%o47)                            [        ]
                                  [ 13  14 ]
```

```
(%i48) addmatrices('max,m1,m2,m1+m2);

                                    [ 8   10 ]
(%o48)                              [        ]
                                    [ 22  24 ]
```

5.7.2 Matrices particulières

• La fonction *diag_matrix*

La fonction **diag_matrix** construit une matrice diagonale à partir des valeurs passées en argument. Par exemple :

```
(%i17) diag_matrix(1,2,3);

                                    [ 1  0  0 ]
                                    [         ]
(%o17)                              [ 0  2  0 ]
                                    [         ]
                                    [ 0  0  3 ]
```

• La fonction *identfor*

La fonction **identfor** remplace les éléments diagonaux de la matrice M par des 1 et les autres par des 0. La fonction **zerofor** remplace tous les éléments de la matrice M par des zéros. Par exemple :

```
(%i29) identfor(M);

                                    [ 1  0  0 ]
                                    [         ]
(%o29)                              [ 0  1  0 ]
                                    [         ]
                                    [ 0  0  1 ]

(%i30) zerofor(M);

                                    [ 0  0  0 ]
                                    [         ]
(%o30)                              [ 0  0  0 ]
                                    [         ]
                                    [ 0  0  0 ]
```

• La fonction *zeromatrixp*

La fonction **zeromatrixp** permet de tester si tous les termes d'une matrice sont nuls. Par exemple :

```
(%i28) zeromatrixp(zerofor(M));

(%o28)                               true

(%i29) zeromatrixp(identfor(M));
```

```
(%o29)                                false
```

• La fonction *toeplitz*

La fonction **toeplitz** construit une matrice de Toeplitz : le premier argument donne la liste des valeurs pour la première colonne et optionnellement un second argument donne la liste des valeurs pour la première ligne (à l'exception du premier terme). Par exemple :

```
(%i4) toeplitz([a,b,c,d]);
```

$$(\%o4)\quad \begin{bmatrix} a & b & c & d \\ b & a & b & c \\ c & b & a & b \\ d & c & b & a \end{bmatrix}$$

```
(%i5) toeplitz([a,b,c,d],[r,s,t,u]);
```

$$(\%o5)\quad \begin{bmatrix} a & s & t & u \\ b & a & s & t \\ c & b & a & s \\ d & c & b & a \end{bmatrix}$$

• La fonction *hankel*

La fonction **hankel** construit une matrice de Hankel. Le premier argument donne les valeurs de la première ligne (ou colonne puisque la matrice est symétrique). Un second argument optionnel donne le nom les valeurs pour compléter les bloc inférieur droit. Le plus simple est de voir son fonctionnement sur des exemples :

```
(%i33) hankel([a,b,c,d]);
```

$$(\%o33)\quad \begin{bmatrix} a & b & c & d \\ b & c & d & 0 \\ c & d & 0 & 0 \\ d & 0 & 0 & 0 \end{bmatrix}$$

```
(%i34) hankel([a,b,c,d],[x,y,z,t]);
```

$$(\%o34)\quad \begin{bmatrix} a & b & c & d \\ b & c & d & y \\ c & d & y & z \\ d & y & z & t \end{bmatrix}$$

• La fonction *hilbert_matrix*

La fonction **hilbert_matrix** construit la matrice de Hilbert de taille n, de terme général $\frac{1}{i+j-1}$.
On obtient par exemple :

```
(%i18) hilbert_matrix (4);
```

```
                                  [    1  1  1 ]
                                  [ 1  -  -  - ]
                                  [    2  3  4 ]
                                  [            ]
                                  [ 1  1  1  1 ]
                                  [ -  -  -  - ]
                                  [ 2  3  4  5 ]
(%o18)                            [            ]
                                  [ 1  1  1  1 ]
                                  [ -  -  -  - ]
                                  [ 3  4  5  6 ]
                                  [            ]
                                  [ 1  1  1  1 ]
                                  [ -  -  -  - ]
                                  [ 4  5  6  7 ]
```

• La fonction *vandermonde_matrix*

La fonction **vandermonde_matrix** construit une matrice de Vandermonde, c'est-à-dire une matrice carrée dont les lignes sont de la forme $1\ x_i\ x_i^2 \ldots x_i^{n-1}$. Par exemple :

```
(%i14) vandermonde_matrix([a,b,c]);
```

```
                                  [        2 ]
                                  [ 1  a   a  ]
                                  [          ]
(%o14)                            [        2 ]
                                  [ 1  b   b  ]
                                  [          ]
                                  [        2 ]
                                  [ 1  c   c  ]
```

• La fonction *polytocompanion*

La fonction **polytocompanion** renvoie la matrice compagnon associée à un polynôme :

```
(%i39) polytocompanion(a*x^2+b*x+c,x);
```

```
                                  [      c ]
                                  [ 0  - - ]
                                  [      a ]
(%o39)                            [        ]
                                  [      b ]
                                  [ 1  - - ]
                                  [      a ]
```

• La fonction *jacobian*

La fonction **jacobian** calcule le jacobien d'une fonction de plusieurs variables. Le second argument indique les variables par rapport auxquelles il faut dériver. Par exemple :

```
(%i28) jacobian([x^2+y^2, x/y],[x,y]);

                                  [ 2 x  2 y  ]
                                  [           ]
(%o28)                            [  1     x  ]
                                  [  -   - -- ]
                                  [  y      2 ]
                                  [         y ]
```

• La fonction *hessian*

La fonction **hessian** calcule la matrice hessienne d'une fonction de plusieurs variables. Le second argument indique les variables par rapport auxquelles il faut dériver. Par exemple :

```
(%i31) hessian(x/y,[x,y]);

                                  [          1  ]
                                  [  0     - -- ]
                                  [           2 ]
                                  [          y  ]
(%o31)                            [             ]
                                  [   1    2 x  ]
                                  [ - --   ---  ]
                                  [    2    3   ]
                                  [   y    y    ]
```

5.7.3 Sous-espaces

• La fonction *nullspace*

```
(%i19) nullspace(M);

                                       [ - 3 ]
                                       [     ]
(%o19)                            span([  6  ])
                                       [     ]
                                       [ - 3 ]

(%i20) nullity(M);

(%o20)                                  1
```

• La fonction *orthogonal_complement*

```
(%i22) orthogonal_complement(transpose([-1,2,-1]));

                                  [ - 2 ]  [  0  ]
                                  [     ]  [     ]
(%o22)                       span([ - 1 ], [ - 1 ])
                                  [     ]  [     ]
                                  [  0  ]  [ - 2 ]
```

• La fonction *columnspace*

```
(%i42) columnspace(M);

                                    [ 1 ]  [ 2 ]
                                    [   ]  [   ]
(%o42)                         span([ 4 ], [ 5 ])
                                    [   ]  [   ]
                                    [ 7 ]  [ 8 ]
```

5.7.4 Fonctionnelles matricielles

La fonction **mat_trace** renvoie la trace d'une matrice :

```
(%i31) mat_trace(vandermonde_matrix([a,b,c]));

(%o31)                              c  + b + 1
```

La fonction **matrix_size** renvoie une liste de deux éléments correspondant au nombre de lignes et de colonnes de la matrice respectivement :

```
(%i33) matrix_size(ident(3));

(%o33)                                [3, 3]
```

La fonction **rank** calcule le rang d'une matrice. Par exemple :

```
(%i38) rank(vandermonde_matrix([1,1,2]));

(%o38)                                   2
```

• La fonction *mat_norm*

La fonction **mat_norm** calcule plusieurs types de normes matricielles. Le second argument peut prendre les valeurs 1, *inf*, *frobenius* (correspondant aux normes $||\ ||_1$, $||\ ||_\infty$ et $||\ ||_F$ respectivement) :

```
(%i70) mat_norm (M, frobenius);

(%o70)                              sqrt(285)

(%i71) mat_norm (M, 1);
```

```
(%o71)                                  18

(%i72) mat_norm (M, inf);

(%o72)                                  24
```

• La fonction *mat_cond*

La fonction **mat_cond** calcule le conditionnement d'une matrice pour une norme matricielle donnée. Le second argument peut prendre les valeurs 1 ou inf pour désigner les normes $||\ ||_1$ et $||\ ||_\infty$ respectivement. Le conditionnement est la quantité $||A|| \times ||A^{-1}||$. Par exemple :

```
(%i24) H: hilbert_matrix(4)$

(%i25) mat_cond(M,1);

(%o25)                                  28375

(%i26) mat_cond(M,inf);

(%o26)                                  28375

(%i27) mat_cond(M,frobenius);

                          4 sqrt(100517) sqrt(1671229)
(%o27)                    ----------------------------
                                      105

(%i28) %,numer;

(%o28)                          15613.79355964381
```

5.7.5 Matrices par blocs

• La fonction *blockmatrixp*

La fonction **blockmatrixp** permet de tester si une matrice est une matrice découpée en blocs. Elle renvoie la valeur TRUE si et seulement si tous les termes de la matrice sont eux-mêmes des matrices. Par exemple :

```
(%i24) B: matrix([matrix([2]),matrix([-1,3])],
                 [matrix([-1],[0]),matrix([1,2],[3,4])]);

                            [   [ 2 ]    [ - 1  3 ] ]
                            [                       ]
(%o24)                      [ [ - 1 ]    [ 1  2 ]   ]
                            [ [     ]    [      ]   ]
                            [ [  0  ]    [ 3  4 ]   ]

(%i25) blockmatrixp(B);

(%o25)                                  true
```

```
(%i26) blockmatrixp(M);

(%o26)                                  false
```

• La fonction *mat_unblocker*

La fonction **mat_unblocker** supprime le découpage en blocs au premier niveau. Par exemple :

```
(%i27) mat_unblocker(B);

                                  [  2   - 1  3 ]
                                  [             ]
(%o27)                            [ - 1   1   2 ]
                                  [             ]
                                  [  0    3   4 ]
```

Pour supprimer un découpage par blocs à tous les niveaux de profondeur, on utilise la fonction **mat_fullunblocker**.

5.7.6 Décompositions matricielles

• La fonction *cholesky*

La fonction **cholesky** calcule la décomposition de Cholesky d'une matrice symétrique définie positive S, c'est-à-dire la matrice triangulaire inférieure L telle que $S = L \times {}^tL$. Par exemple :

```
(%i20) S: matrix([1,-1,-2],[-1,5,4],[-2,4,14]);

                                  [  1   - 1  - 2 ]
                                  [               ]
(%o20)                            [ - 1   5    4  ]
                                  [               ]
                                  [ - 2   4   14  ]

(%i21) L: cholesky(S);

                                    [  1   0  0 ]
                                    [           ]
(%o21)                              [ - 1  2  0 ]
                                    [           ]
                                    [ - 2  1  3 ]
```

• La fonction *lu_factor*

La fonction **lu_factor** calcule la décomposition LU d'une matrice carrée. Elle renvoie le résultat sous une forme compacte où figurent à la fois la matrice triangulaire inférieure L et la matrice triangulaire supérieure U. Prenons par exemple la matrice B suivante :

```
(%i56) B: matrix([2, 1, 1],[4, -6, 0],[-2,7,2]);
```

```
                                  [  2    1   1 ]
                                  [             ]
(%o56)                            [  4   - 6  0 ]
                                  [             ]
                                  [ - 2   7   2 ]
```

La décomposition LU de cette matrice renvoie le résultat suivant :

```
(%i57) lu: lu_factor(B);

                    [  2    1    1  ]
                    [               ]
(%o57)             [[  2   - 8  - 2 ], [1, 2, 3], generalring]
                    [               ]
                    [ - 1  - 1   1  ]
```

Pour séparer les termes L et U, on utilise la fonction **get_lu_factors** comme ceci :

```
(%i58) get_lu_factors(lu);

                  [ 1  0  0 ]  [  1    0   0 ]  [ 2   1    1  ]
                  [         ]  [             ]  [              ]
(%o58)           [[ 0  1  0 ], [  2    1   0 ], [ 0  - 8  - 2 ]]
                  [         ]  [             ]  [              ]
                  [ 0  0  1 ]  [ - 1  - 1  1 ]  [ 0   0    1  ]
```

Un deuxième argument permet de spécifier le domaine dans lequel sont effectués les calculs. Les valeurs possibles sont *generalring* (valeur par défaut), *floatfield*, *complexfield*, *crering*, *rationalfield*, *runningerror*, *noncommutingring*. En voici deux exemples :

```
(%i62) lu: lu_factor(B, floatfield);

            [  0.5    4.0   1.0 ]
            [                   ]
(%o62)     [[  4.0   - 6.0  0.0 ], [2, 1, 3], floatfield, 11.0, 33.0]
            [                   ]
            [ - 0.5   1.0   1.0 ]

(%i67) lu: lu_factor(B, rationalfield);

              [  1          ]
              [  -    4   1 ]
              [  2          ]
              [             ]
(%o67)       [[  4   - 6  0 ], [2, 1, 3], rationalfield, 11.0, 33.0]
              [             ]
              [   1         ]
              [ - -   1   1 ]
              [   2         ]
```

6 Polynômes

6.1 Représentation des polynômes

Dans Maxima, un polynôme est représenté de deux manières différentes :
– comme une expression algébrique usuelle ;
– comme une liste utilisant le format CRE.
Sous forme algébrique, le polynôme $5x^2 - 3x + 7$ serait déclaré comme ceci :

```
(%i1) p: 5*x^2-3*x+7;

                                    2
(%o1)                          5 x  - 3 x + 7
```

La forme CRE (abréviation de *Canonical Rational Expression*) est un format de représentation interne pour des expressions à plusieurs variables. Les variables doivent être ordonnées par ordre d'importance. La représentation CRE d'un polynôme consiste en une liste récursive comportant le nom de la variable principale suivi de paires d'expressions correspondant à chaque terme du polynôme. Le premier terme de chaque paire représente l'exposant de la variable principale et le second terme est le coefficient correspondant qui peut être un simple nombre ou lui-même un polynôme en une autre variable exprimé dans le format CRE.

La forme CRE du polynôme $5x^2-3x+7$ est ainsi `(x 2 5 1 -3 0 7)`. Celle du polynôme $5x^2 + 7$ est `(x 2 5 0 7)`.

Pour le polynôme $x^2 - 2xy + 3y^2 + 5x - 7$, la forme CRE sera

```
(x 2 1 1 (y 1 -2 0 5) 0 (y 2 3 0 -7))
```

si on suppose que x est la variable principale et y une variable secondaire. Dans le cas contraire, si y est la variable principale et x une variable secondaire, ce serait

```
(y 2 3 1 (x 1 -2) 0 (x 2 1 1 5 0 -7)).
```

L'ordre des variables peut être spécifié par la fonction **ratvars**. Par défaut, Maxima adopte l'ordre alphabétique des variables.

Maxima indique qu'une expression est représentée en interne sous la forme CRE en ajoutant le symbole `/R/` en début de ligne.

Les expressions rationnelles, quotients de deux polynômes, ont aussi une représentation de type CRE qui contient essentiellement une liste des variables ordonnées et une paire de polynômes.

La fonction **rat**, qui sert à simplifier des expressions rationnelles, a pour effet de convertir l'expression initiale au format CRE. C'est ce qu'indique le symbole `/R/` dans l'exemple suivant :

```
(%i1)  rat(x^2+3*x-1);

                                    2
(%o1)/R/                       x  + 3 x - 1
```

La fonction **showratvars** renvoie une liste des variables CRE présentes dans une expression.

La variable *ratvars* est la liste des variables passées dans l'appel le plus récent à la fonction du même nom. Si on invoque cette fonction sans arguments, la liste est vidée.

L'opération inverse de conversion d'une expression CRE à la forme générale est effectuée par la fonction **ratdisrep**. La fonction **totaldisrep** fait la même conversion sur toutes les sous-expressions d'une expression.

La fonction **ratp** permet de tester si une expression est au format CRE ou pas.

6.2 Eléments d'un polynôme

Les fonctions **num** et **denom** renvoient respectivement le numérateur et le dénominateur d'une expression rationnelle.

Les fonctions **hipow** et **lopow** renvoient respectivement l'exposant explicite le plus haut et le plus bas qui figure dans une expression. La syntaxe est :

```
hipow (expr, x)
lopow (expr, x)
```

où x désigne la variable considérée. Ces deux commandes n'effectuent aucune simplification comme on peut le voir sur les exemples suivants :

```
(%i5) p: (x^3-x)/(x^4+x^2);
                                        3
                                       x  - x
(%o5)                                  -------
                                        4    2
                                       x  + x

(%i6) lopow(p,x);

(%o6)                                    1

(%i7) hipow(p,x);

(%o7)                                    4
```

• La fonction *powers*

La fonction **powers** s'applique à un polynôme et non pas une fraction rationnelle. Elle renvoie la liste des puissances qui sont présentes dans le polynôme. Cette fonction requiert le chargement du package **powers**. Par exemple :

```
(%i11) load(powers)$
(%i12) powers(x^4+x^2,x);

(%o12)                                [4, 2]
```

• La fonction *coeff*

La fonction **coeff** renvoie le coefficient du terme de degré n dans un polynôme. La syntaxe est :

```
coeff (expr, x, n)
coeff (expr, x)
```

Si n n'est pas spécifié, il vaut 1 par défaut. Cette fonction n'applique aucune simplification avant de rechercher le coefficient. Par exemple :

```
(%i15) p: x^2*y+3*x^2-x*y+(x+1)*(x+3);
                          2               2
(%o15)                   x  y - x y + 3 x  + (x + 1) (x + 3)
```

```
(%i16) coeff(p, x, 2);

(%o16)                                  y + 3
```

• La fonction *ratcoef*

La fonction **ratcoef** au contraire commence par développer et simplifier le polynôme. Par exemple, en reprenant l'exemple précédent :

```
(%i17) ratcoeff(p, x, 2);

(%o17)                                  y + 4
```

• La fonction *bothcoef*

La fonction **bothcoef** renvoie une liste de deux expressions : celle qui est en coefficient de la variable spécifée et celle qui comporte les éléments restants. Typiquement, dans $ax + b$, elle renvoie $[a, b]$. Par exemple :

```
(%i26) bothcoef(y*x+3*x+2,x);

(%o26)                [y + 3, - x (y + 3) + x y + 3 x + 2]
```

• La fonction *polydecomp*

La fonction **polydecomp** s'applique à un polynôme en x et renvoie une liste de polynômes qui redonnent l'expression de départ par composition. Par exemple :

```
(%i29) p: expand( (x-1)^3 );

                                3       2
(%o29)                         x  - 3 x  + 3 x - 1

(%i30) polydecomp(p,x);

                                     3
(%o30)                              [x , x - 1]
```

6.3 Opérations sur les polynômes

• La fonction *eliminate*

La fonction **eliminate** élimine des variables entre plusieurs équations polynômiales à plusieurs variables (ou expressions polynômiales supposées égales à 0). Par exemple :

```
(%i23) p1: x^2+y^2+z^2;

                                   2    2    2
(%o23)                            z  + y  + x

(%i24) p2: x+3*y+z=1;
```

```
(%o24)                          z + 3 y + x = 1

(%i25) p3: z-2*y+3*x=1;

(%o25)                        z - 2 y + 3 x = 1

(%i26) eliminate([p1,p2,p3],[y,z]);

                              4          3         2
(%o26)            [5 (1620 x  - 1608 x  + 608 x  - 92 x + 5)]

(%i27) eliminate([p1,p2,p3],[y]);

           2                       2
(%o27) [5 z  + (6 x - 2) z + 13 x  - 6 x + 1,
                2                     2
            10 z  + (2 x - 2) z + 10 x  - 2 x + 1]
```

• La fonction *ratdiff*

La fonction **ratdiff** calcule la dérivée, par rapport à une des variables, d'un polynôme ou d'une expression rationnelle. Elle est plus rapide que la fonction **diff**. Si l'expression est au format CRE, la dérivée le sera aussi. Autrement elle est renvoyée dans le format général.

• La fonction *fasttimes*

La fonction **fasttimes** effectue des produits de polynômes en utilisant un algorithme rapide d'ordre $\max(n_1, n_2)^{1.585}$. Il faut que les deux polynômes soient au format CRE. Par exemple :

```
(%i37) p1: x^2+y^2+z^2;

                                  2    2    2
(%o37)                           z  + y  + x

(%i38) p2: z + 3*y + x;

(%o38)                            z + 3 y + x

(%i39) fasttimes(rat(p1),rat(p2));

           3              2     2    2          3      2      2      3
(%o39)/R/ z  + (3 y + x) z  + (y  + x ) z + 3 y  + x y  + 3 x  y + x
```

• La fonction *polymod*

La fonction **polymod** applique une réduction modulo m aux termes d'un polynôme à coefficients entiers. Typiquement, les valeurs utilisées pour m sont des nombres premiers. Si on prend un nombre premier p, la réduction est appliquée de manière symétrique en renvoyant des valeurs entre $-(p-1)/2$ et $(p-1)/2$. Par exemple :

```
(%i45) p: 5*x^3+2*x^2-x+11;

                                  3      2
(%o45)                       5 x  + 2 x  - x + 11
```

```
(%i46) polymod(p,2);

                                    3
(%o46)                             x  + x + 1

(%i47) polymod(p,3);

                                    3    2
(%o47)                           - x  - x  - x - 1
```

La valeur par défaut utilisée pour cette réduction peut être spécifiée au moyen de la variable globale *modulus*. Dans ce cas, on peut ne pas préciser le deuxième argument de la fonction. Par exemple :

```
(%i59) modulus;

(%o59)                                false

(%i60) modulus: 5;

(%o60)                                 5

(%i61) polymod(p);

                                     2
(%o61)                            2 x  - x + 1
```

On supprime ce mécanisme en refixant *modulus* à la valeur FALSE.

• La fonction *resultant*

La fonction **resultant** calcule le résultant de deux polynômes p_1 et p_2 en éliminant la variable spécifiée. La syntaxe est :

```
resultant (p_1, p_2, x)
```

Le résultant est le déterminant de la matrice de Sylvester des coefficients de x dans p_1 et p_2. Il s'annule si et seulement si p_1 et p_2 n'ont pas de facteurs communs. Par exemple :

```
(%i62) resultant( x^2+x,x-1,x);

(%o62)                                 2

(%i63) resultant( x^2+x,x+1,x);

(%o63)                                 0
```

• La fonction *bezout*

La fonction **bezout** renvoie une matrice dont le déterminant est égal au résultant. Par exemple :

```
(%i64) bezout( x^2+x,x-1,x);

                                  [ - 1  - 1 ]
(%o64)                            [          ]
                                  [  1   - 1 ]
```

```
(%i66) bezout( a*x^2+b*x,c*x+d,x);

                                  [ a d  b d ]
(%o66)                            [          ]
                                  [  c    d  ]
```

• La fonction *ratweight*

La fonction **ratweight** applique un poids w_i à la variable x_i de telle sorte qu'un terme est remplacé par 0 si son poids est supérieur à un certain niveau. Ce niveau est fixé au moyen de la variable *ratwtlvl*. Le poids d'un terme composite est la somme des poids des variables qui le composent compte-tenu de leur exposant : par exemple, le poids de $x_1^3 x_2^2$ est $3w_1 + 2w_2$. La syntaxe est :

```
ratweight (x_1, w_1,..., x_n, w_n)
```

La variable est fixée à FALSE par défaut pour désactiver ce mécanisme. La réduction est effectuée seulement lorsqu'on multiplie ou élève à une certaine puissance des expressions dans le format CRE. Par exemple :

```
(%i1)

                                   3    2
(%o1)                             x  + x  + x + 1

(%i2) p2: x^4+x^3+x^2+x+1;

                                 4    3    2
(%o2)                           x  + x  + x  + x + 1

(%i3) ratweight(x,3);

(%o3)                                [x, 3]

(%i4) ratwtlvl: 7;

(%o4)                                  7

(%i5) expand(p1*p2);

                  7      6      5      4      3      2
(%o5)            x  + 2 x  + 3 x  + 4 x  + 4 x  + 3 x  + 2 x + 1

(%i6) expand(rat(p1)*rat(p2));

                                     2
(%o6)                             3 x  + 2 x + 1
```

6.4 Facteurs et racines

• La fonction *factor*

La fonction **factor** effectue des factorisations de polynômes en facteurs irréductibles sur les nombres entiers. Par exemple :

```
(%i5) p: x^3+6*x^2+11*x+6;

                                3       2
(%o5)                          x  + 6 x  + 11 x + 6

(%i6) factor(p);

(%o6)                        (x + 1) (x + 2) (x + 3)
```

Appliquée à un nombre entier, cette fonction a pour effet de le décomposer en un produit facteurs entiers. Par exemple :

```
(%i13) factor(126);

                                          2
(%o13)                                2 3  7
```

• La fonction *sqfr*

La fonction **sqfr** (abréviation de *square-free*) est analogue à la fonction **factor** mais elle produit des sous-expressions qui ne contiennent que des facteurs de degré 1. Par exemple, comparer :

```
(%i24) sqfr (4*x^4 + 4*x^3 - 3*x^2 - 4*x - 1);

                                       2   2
(%o24)                          (2 x + 1)  (x  - 1)

(%i25) factor (4*x^4 + 4*x^3 - 3*x^2 - 4*x - 1);

                                                        2
(%o25)                     (x - 1) (x + 1) (2 x + 1)
```

Ici la fonction **sqfr** a produit le terme $(x^2 - 1)$ qui ne se décompose pas avec des facteurs de degré 2 ou plus.

• La fonction *factorout*

La fonction **factorout** transforme une expression en une somme de produits factorisés selon certaines variables. La syntaxe est :

```
    factorout (expr, x_1, x_2,...)
```

Elle a pour effet de réarranger l'expression en une somme de termes de la forme $f(x_1, x_2, \dots) * g$ où g ne contient pas les variables x_1, x_2, etc. Par exemple :

```
(%i19) expr: a*x^2+y*x^2+3*x*y+b*x*y^2+2*x-y+1;

                      2    2                     2
(%o19)           b x y  + x  y + 3 x y - y + a x  + 2 x + 1

(%i20) factorout(expr,x);

                      2     2                    2
(%o20)           b x y  + (x  + 3 x - 1) y + a x  + 2 x + 1
```

```
(%i21) factorout(expr,y);

                        2                   2           2
(%o21)           b x y  + x (3 y + 2) + x  y - y + a x  + 1
```

• La fonction *factorsum*

La fonction **factorsum** réarrange les termes d'une expression polynômiale pour les regrouper en sous-expressions factorisables. Par exemple :

```
(%i22) expr: expand( ((a+b)^2 + (c-d)^2)*(u+v)  );

       2                2      2                2      2                2
(%o22) d  v - 2 c d v + c  v + b  v + 2 a b v + a  v + d  u - 2 c d u + c  u
                                     2               2
                                  + b  u + 2 a b u + a  u

(%i23) factorsum(expr);

                                  2          2
(%o23)                   ((d - c)  + (b + a) ) (v + u)
```

• La fonction *allroots*

Pour calculer les racines d'un polynôme, on peut utiliser l'une des deux fonctions suivantes :

- la fonction **allroots** qui trouve toutes les racines réelles ou complexes d'un polynôme ou d'une équation polynômiale à une variable ;
- la fonction **realroots** qui ne renvoie que les racines réelles compte-tenu d'une certaine tolérance.

Par exemple :

```
(%i13) eqn: (x+1)^4=16*(2-x);

                                    4
(%o13)                       (x + 1)  = 16 (2 - x)

(%i14) allroots(eqn);

(%o14) [x = 1.0, x = 2.699992796007314 %i - .4252569261129805,
       x = - 2.699992796007314 %i - .4252569261129805,
       x = - 4.149486147774039]
```

Si la variable *polyfactor* est fixée à la valeur TRUE, la fonction **allroots** a pour effet de factoriser le polynôme en facteurs réels de degré 1 ou 2 :

```
(%i15) polyfactor: true;

(%o15)                                true

(%i16) allroots(eqn);

(%o16) 1.0 (x - 1.0) (x + 4.149486147774039)
                     2
                  (x  + 0.850513852225961 x + 7.470804551698454)
```

Si le polynôme est à coefficients complexes, la factorisation se fait en facteurs complexes de degré 1. La fonction **allroots** n'utilise pas le même algorithme selon que le polynôme est à coefficients réels ou complexes.

Une variante de cette fonction est la fonction **bfallroots** qui effectue les calculs avec double précision (en *big floats*).

• La fonction *realroots*

La fonction **realroots** admet un second argument qui spécifie une tolérance pour la précision du calcul. Elle ne calcule que les racines réelles et écrit la multiplicité des différentes racines dans la variable *multiplicities*. Par exemple :

```
(%i17) p: expand( (x-2)^2*(x+1)^3 );

                          5    4      3    2
(%o17)                   x  - x  - 5 x  + x  + 8 x + 4

(%i18) realroots(p, 1e-5);

(%o18)                          [x = - 1, x = 2]

(%i19) multiplicities;

(%o19)                                [3, 2]
```

6.5 Division polynômiale

• La fonction *divide*

La fonction **divide** effectue la division euclidienne de deux polynômes. La syntaxe est :

```
divide (p_1, p_2, x_1,..., x_n)
```

Le calcul est fait par rapport à la variable x_n. Les autres variables sont utilisées comme dans la fonction **ratvars**. Si x_n est absente, c'est la variable x_{n-1} qui est utilisée etc. La valeur de retour est une liste de deux éléments : le premier est le quotient, le second est le reste.

Ces deux polynômes peuvent aussi être obtenus séparément au moyen des fonction **quotient** et **remainder**.

Par exemple :

```
(%i4) p1: x^2+x+1;

                                    2
(%o4)                              x  + x + 1

(%i5) p2: x+2;

(%o5)                                 x + 2

(%i6) divide(p1,p2,x);

(%o6)                              [x - 1, 3]

(%i7) quotient(p1,p2,x);
```

```
(%o7)                                  x - 1

(%i8) remainder(p1,p2,x);

(%o8)                                    3
```

• La fonction *gcd*

La fonction **gcd** calcule le plus grand commun diviseur entre deux polynômes. La syntaxe est :

```
gcd (p_1, p_2, x_1,...)
```

Par exemple :

```
(%i10) p1: x^2+3*x+2;

                                    2
(%o10)                             x  + 3 x + 2

(%i11) p2: x^2-2*x-3;

                                    2
(%o11)                             x  - 2 x - 3

(%i12) gcd(p1,p2,x);

(%o12)                                 x + 1
```

Il existe plusieurs algorithmes pour effectuer le calcul. On peut les spécifier au moyen de la variable gcd : les valeurs possibles sont *ez*, *subres*, *red* ou *spmod* qui correspondent respectivement aux algorithmes *ezgcd*, sous-résultant, réduit ou modulaire.

Il y a d'autres fonctions disponibles dans le package **grobner**, en particulier la fonction **poly_gcd**.

• La fonction *ezgcd*

La fonction **ezgcd** permet de traiter plus de deux polynômes. Sa syntaxe est :

```
ezgcd (p_1, p_2, p_3,...)
```

Elle renvoie une liste dont le premier élément est le plus grand commun diviseur et les suivants sont les quotients des polynômes par ce PGCD. Par exemple

```
(%i15) p3: x^2-1;

                                    2
(%o15)                             x  - 1

(%i16) ezgcd(p1,p2,p3);

(%o16)                  [x + 1, x + 2, x - 3, x - 1]
```

• La fonction *content*

La fonction **content** calcule le PGCD des termes d'un polynôme à coefficients entiers. La syntaxe est :

```
content (p_1, x_1,..., x_n)
```

Elle renvoie une liste dont le premier élément est le plus grand commun diviseur et le suivant est le quotient du polynôme par ce PGCD. Par exemple :

```
(%i17) p: 36*x^2+120*x+18;
                                    2
(%o17)                         36 x  + 120 x + 18

(%i18) content(p,x);
                                      2
(%o18)                        [6, 6 x  + 20 x + 3]
```

• La fonction *gcdex*

La fonction **gcdex** renvoie une liste de la forme $[a, b, u]$ où u est le plus grand commun diviseur de deux polynômes p et q et où a et b sont les entiers tels que

$$u = ap + bq.$$

Par exemple :

```
(%i21) gcdex(p1,p2);
(%o21)/R/                          [1, - 1, 5 x + 5]

(%i22) p1-p2;
(%o22)                                 5 x + 5
```

• La fonction *gcfactor*

La fonction **gcfactor** factorise un entier en un produit de nombres de Gauss, c'est-à-dire de nombres complexes à coefficients entiers. Par exemple :

```
(%i37) gcfactor(5);
(%o37)                       - %i (1 + 2 %i) (2 + %i)

(%i38) gcfactor(100);
                                    4            2          2
(%o38)                    (1 + %i)  (1 + 2 %i)  (2 + %i)
```

• La fonction *gfactor*

La fonction **gfactor** factorise un polynôme à coefficients entiers sur l'ensemble des entiers de Gauss. Par exemple :

```
(%i50) gfactor(x^2+1);
(%o50)                           (x - %i) (x + %i)
```

```
(%i51) gfactor(x^3+x^2+x+1);

(%o51)                        (x + 1) (x - %i) (x + %i)
```

• La fonction *gfactorsum*
La fonction **gfactorsum** est analogue à la fonction **factorsum** mais elle s'appuie sur **gfactor** au lieu de **factor**.

6.6 Simplification d'expressions rationnelles

• La fonction *rat*
La fonction **rat** a été mentionnée à la section 6.1. Elle convertit des expressions rationnelles au format CRE et opère au passage des simplifications en développant et regroupant tous les termes, en réduisant au même dénominateur et en divisant le numérateur et le dénominateur par leur plus grand commun diviseur. Par exemple :

```
(%i62) q: (x+2*y)^2/(x+y-1)+(y-x)^2/(x-y+1);

                                    2            2
                             (2 y + x)      (y - x)
(%o62)                       ---------- + ----------
                             y + x - 1    - y + x + 1

(%i63) rat(q);

                      3             2          2                3
                   3 y  + (x - 3) y  + (- 2 x  - 6 x) y - 2 x
(%o63)/R/          ------------------------------------------
                                 2          2
                                y  - 2 y - x  + 1
```

La fonction **rat** convertit aussi tous les nombres en virgule flottante sous forme de fractions.

```
(%i68) r: (2.4*x+3.2*y)^2/(x-0.5*y)+2*x+y;

                                  2
                          (3.2 y + 2.4 x)
(%o68)                    --------------- + y + 2 x
                             x - 0.5 y

(%i69) rat(r);

rat: replaced -0.5 by -1/2 = -0.5

rat: replaced 2.4 by 12/5 = 2.4

rat: replaced 3.2 by 16/5 = 3.2
                                 2                    2
                               487 y  + 768 x y + 388 x
(%o69)/R/                  - ------------------------
                                    25 y - 50 x
```

Les fonctions **ratnumer** et **ratdenom** procèdent de manière analogue et retournent respectivement le numérateur et le dénominateur :

```
(%i2) ratnumer(r);

rat: replaced -0.5 by -1/2 = -0.5
rat: replaced 2.4 by 12/5 = 2.4
rat: replaced 3.2 by 16/5 = 3.2
                                  2                    2
(%o2)/R/                   - 487 y  - 768 x y - 388 x

(%i3) ratdenom(r);

rat: replaced -0.5 by -1/2 = -0.5
rat: replaced 2.4 by 12/5 = 2.4
rat: replaced 3.2 by 16/5 = 3.2
(%o3)/R/                             25 y - 50 x
```

• La fonction *ratexpand*

La fonction **ratexpand** opère le même genre de transformation que la fonction **rat** mais elle fractionne ensuite le numérateur en chacun de ses termes divisé par le dénominateur commun. D'autre part, le résultat qu'elle renvoie est dans la forme générale et non pas dans le format CRE, même si l'expression originale était en format CRE. L'exemple suivant permet de comparer le comportement des deux fonctions :

```
(%i83) q: (x-1)/(x+y) + (y-1)/(y-x);

                                   x - 1   y - 1
(%o83)                             ----- + -----
                                   y + x   y - x

(%i84) rat(q);

                                 2                   2
                                y  + (2 x - 2) y - x
(%o84)/R/                       --------------------
                                       2    2
                                      y  - x

(%i85) ratexpand(q);

                          2                              2
                         y          2 x y        2 y     x
(%o85)                 ------- + ------- - ------- - -------
                        2    2     2    2     2    2    2    2
                       y  - x     y  - x     y  - x    y  - x
```

• La fonction *ratsimp*

La syntaxe de la fonction **ratsimp** peut prendre deux formes :

```
ratsimp (expr)
ratsimp (expr, x_1,..., x_n)
```

Elle simplifie l'expression et toutes les sous-expressions et renvoie un quotient de deux polynômes sous forme récurrente : les coefficients de la variable principale sont des polynômes en les autres variables qui sont eux-mêmes exprimés sur le même modèle. Par exemple :

```
(%i98) q: (x-z*y)/(x+y) + (y*x-z)/(y-x);
```

$$(\%o98)\qquad \frac{x - y\,z}{y + x} + \frac{x\,y - z}{y - x}$$

```
(%i99) ratsimp(q);
```

$$(\%o99)\qquad -\frac{(y^2 + (1 - x)\,y + x)\,z - x\,y^2 + (-x^2 - x)\,y + x^2}{y^2 - x^2}$$

```
(%i100) ratsimp(q,x);
```

$$(\%o100)\qquad -\frac{x\,((y - 1)\,z + y^2 + y) + (-y^2 - y)\,z + x^2\,(y - 1)}{x^2 - y^2}$$

```
(%i101) ratsimp(q,x,y);
```

$$(\%o101)\qquad \frac{y\,(x\,(z + 1) - z + x^2) - x\,z + y^2\,(x - z) - x^2}{y^2 - x^2}$$

```
(%i5) ratsimp(q,x,z);
```

$$(\%o5)\qquad -\frac{(-y^2 - y + x\,(y - 1))\,z + x\,(y^2 + y) + x^2\,(y - 1)}{x^2 - y^2}$$

• La fonction *fullratsimp*

La fonction **fullratsimp** applique **ratsimp** de manière répétitive jusqu'à ce que l'expression ne puisse plus être modifiée. Par exemple :

```
(%i137) q: (x^(a/2) + 1)^2*(x^(a/2) - 1)^2/(x^a - 1);
```

```
                                  a/2     2   a/2     2
                                 (x    - 1)  (x    + 1)
(%o137)                          ----------------------
                                           a
                                          x  - 1
```

```
(%i138) ratsimp(q);

                                   2 a      a
                                  x    - 2 x  + 1
(%o138)                           ---------------
                                       a
                                      x  - 1
```

```
(%i139) fullratsimp(q);

                                       a
(%o139)                               x  - 1
```

6.7 Substitutions polynômiales

La fonction **ratsubst** a été expliquée à la section 3.2. La fonction **fullratsubst** applique **ratsubst** de manière répétitive jusqu'à ce que l'expression ne puisse plus être modifiée. La fonction **lratsubst** prend en argument une liste d'équations et une expression. La syntaxe est :

```
lratsubst (L, expr)
```

Remarque : ces deux dernières fonctions sont définies dans le package **lrats** qu'il faut charger avant de les utiliser. Par exemple :

```
(%i145) load("lrats")$
(%i146) lratsubst ([a^2 = c, b^2 = d], (a + b)^3);
(%o146)                     b d + 3 a d + (3 b + a) c
```

6.8 Développements limités

Les développements limités sont calculés au moyen de la fonction **taylor**. Maxima utilise une forme étendue de la notation CRE pour représenter des séries de Taylor. Ceci est indiqué par un symbole `/T/` placé après l'étiquette des lignes des sorties.

```
(%i6) g: f / sinh(k*x)^4;

                           3   k x
                          x  %e    sin(w x)
(%o6)                     -----------------
                                 4
                             sinh (k x)
```

```
(%i7) taylor (g, x, 0, 3);

                          2    3   2          2    3   3
         w    w x   (w k  + w ) x    (3 w k  + w ) x
(%o7)/T/ -- + --- - -------------- - ---------------- + . . .
          4    3            4                  3
         k    k          6 k                6 k
```

cos	*acos*
cosh	*acosh*
cot	*acot*
coth	*acoth*
csc	*acsc*
csch	*acsch*
sec	*asec*
sech	*asech*
sin	*asin*
sinh	*asinh*
tan	*atan* et *atan2*
tanh	*atanh*

TABLE 2 – Fonctions trigonométriques et leurs réciproques

7 Trigonométrie

7.1 Fonctions trigonométriques

La plupart des fonctions trigonométriques ont déjà été mentionnées dans le tableau 1 de la page 13. Le tableau 2 en donne la liste complète.

7.2 Expressions trigonométriques

Il existe quelques fonctions qui permettent de simplifier, développer ou réduire des expressions constituées de termes trigonométriques.

• La fonction *trigexpand*

La fonction **trigexpand** développe les cosinus et sinus d'angles multiples en combinaisons de puissances. Par exemple :

```
(%i4) trigexpand(cos(5*x));

                              4            3       2          5
(%o4)           5 cos(x) sin (x) - 10 cos (x) sin (x) + cos (x)
```

• La fonction *trigreduce*

La fonction **trigreduce** fait le contraire de la fonction **trigexpand** : elle essaye de linéariser les expressions polynomiales en cosinus et sinus en les transformant en combinaisons de cosinus et sinus d'angles multiples. Par exemple :

```
(%i5) trigreduce(cos(x)^3-sin(x)^3);

                    sin(3 x)   cos(3 x) + 3 cos(x)   3 sin(x)
(%o5)               -------- + ------------------ - --------
                       4               4               4
```

```
(%i1) trigreduce(cos(x)^2-sin(x)^2);
```

```
                              cos(2 x) + 1   cos(2 x)    1
(%o1)                         ------------ + -------- - -
                                   2            2       2
```

```
(%i2) trigrat(%);

(%o2)                                cos(2 x)
```

• La fonction *trigrat*

La fonction **trigrat** opère sur des fractions rationnelles en cosinus, sinus ou tangentes avec des arguments qui combinent linéairement des variables et des multiples entiers de π/n afin de les linéariser. Par exemple :

```
(%i25) trigrat(cos(a+%pi/3)+cos(a+5*%pi/3));

(%o25)                                cos(a)
```

• La fonction *trigsimp*

La fonction **trigsimp** simplifie des expressions trigonométriques contenant des fonctions **tan**, **sec**, etc. en faisant usage de l'identité de base $\cos^2 + \sin^2 = 1$. Par exemple :

```
(%i24) trigsimp(tan(x)^3/(1+tan(x)^2));

                                        3
                                     sin (x)
(%o24)                               -------
                                     cos(x)
```

7.3 Options trigonométriques

Quelques options permettent d'influer sur le comportement des fonctions de manipulation d'expressions trigonométriques. Elles prennent des valeurs logiques (TRUE ou FALSE). Il s'agit des variables suivantes :

- l'option ***trigexpandplus*** contrôle les expressions trigonométriques dont l'argument est une somme (comme $\cos(x + y)$) ;
- l'option ***trigexpandtimes*** contrôle les expressions trigonométriques dont l'argument est un produit (comme $\cos(2x)$) ;
- l'option ***trigsign*** contrôle le traitement des arguments comportant un signe négatif (comme $\sin(-x)$) ;
- l'option ***triginverses*** contrôle le traitement des expressions contenant des fonction trigonométriques et leur réciproque (comme $\mathrm{atan}(\tan(x))$. Les valeurs possibles sont *all*, TRUE, FALSE. Par exemple :

```
(%i47) triginverses: all;
(%o47)                                all
(%i48) atan(tan(x));
(%o48)                                 x
(%i49) tan(atan(x));
```

```
(%o49)                                    x
```

- l'option *halfangles* contrôle la converion des expressions qui sont fonctions des angles moitié. Par exemple :

```
(%i39) halfangles: true;
(%o39)                                  true
(%i40) assume(a>0, a<2*%pi);
(%o40)                        [a > 0, 2 %pi > a]
(%i41) sin(a/2);
                                  sqrt(1 - cos(a))
(%o41)                            ----------------
                                       sqrt(2)
```

8 Dérivation

8.1 Dérivées et différentielles

• La fonction *diff*

La fonction **diff** renvoie la dérivée ou la différentielle d'une expression par rapport à une ou plusieurs variables. L'argument désignant chaque variable peut être suivi d'un argument entier désignant l'ordre de dérivation (1 par défaut).

La syntaxe la plus complète de cette fonction est

```
diff(expr, x_1, n_1, ..., x_m, n_m)
```

Par exemple :

```
(%i1) f: x^2/y^2;
                                        2
                                       x
(%o1)                                  --
                                        2
                                       y

(%i2) diff(f,x);
                                       2 x
(%o2)                                  ---
                                        2
                                       y

(%i3) diff(f,y);
                                          2
                                       2 x
(%o3)                                - ----
                                          3
                                       y
```

```
(%i4) diff(f,x,2);

                                      2
(%o4)                                 --
                                       2
                                      y

(%i5) diff(f,x,1,y,1);

                                        4 x
(%o5)                                 - ---
                                          3
                                        y

(%i6) diff(f,y,2);

                                         2
                                      6 x
(%o6)                                 ----
                                        4
                                       y
```

Si on utilise la fonction **diff** sans préciser de variable de dérivation, elle calcule la différentielle de l'expression, autrement dit la quantité $df = \dfrac{\partial f}{\partial x}\,dx + \dfrac{\partial f}{\partial y}\,dy$. Par exemple :

```
(%i7) diff(f);

                                                 2
                                  2 x del(x)   2 x  del(y)
(%o7)                             ---------- - -----------
                                       2            3
                                      y            y
```

La fonction **del**, comme on le voit sur le résultat de l'exemple précédent, fournit une notation symbolique pour la différentielle des variables (les termes dx ou dy).

On peut utiliser la fonction **declare** pour indiquer que certaines variables sont constantes. Par exemple :

```
(%i24) declare(a,constant);

(%o24)                                done

(%i25) diff(a*x*y);

(%o25)                    a x del(y) + a y del(x)
```

• La fonction *atvalue*

La fonction **atvalue** attribue une valeur à une expression en un point donné. Elle sert principalement à spécifier des valeurs au bord. Par exemple, pour une fonction inconnue $g(x, y)$, on fixe une valeur initiale telle que $g(0, 1) = a$ au moyen de l'instruction suivante :

```
(%i9) atvalue(g(x,y),[x=1,y=2],a);

(%o9)                                  a

(%i10) g(1,2);

(%o10)                                 a

(%i11) printprops(all,atvalue);

                                  g(1, 2) = a
(%o11)                               done
```

• La fonction *at*

La fonction **at** renvoie la valeur d'une expression en un point donné. Le deuxième argument est une liste d'expressions permettant d'indiquer les coordonnées du point où la valeur est calculée. Par exemple :

```
(%i17) at(f,[x=1,y=2]);

                                       1
(%o17)                                 -
                                       4
```

En reprenant la fonction *g* précédente :

```
(%i21) at((2+f)*g(x,y),[x=1,y=2]);

                                     9 a
(%o21)                               ---
                                      4
```

• La fonction *gradef*

La fonction **gradef** définit (et non pas calcule) les composantes du gradient d'une fonction. La syntaxe peut prendre deux formes différentes :

```
gradef(f(x_1, ..., x_n), g_1, ..., g_m)
gradef(a, x, expr)
```

La première forme attribue les expressions `g_1`,..., `g_m` aux dérivées partielles de la fonction spécifiée par le premier argument. Par exemple :

```
(%i1) gradef(F(x,y),2*x+y,x+2*y);

(%o1)                                F(x, y)

(%i2) printprops(F,gradef);

                              d
                              -- (F(x, y)) = 2 x + y
                              dx
                              d
                              -- (F(x, y)) = x + 2 y
                              dy
(%o2)                                done
```

```
(%i3) diff(F(x,y),x);

(%o3)                                  y + 2 x
```

La deuxième forme définit la dérivée de a par rapport à x comme étant l'expression spécifiée. Cette forme établit en même temps une dépendance de a sur la variable x.

```
(%i5) gradef(a,x,log(x));

(%o5)                                     a

(%i6) dependencies;

(%o6)                                  [a(x)]

(%i7) diff(a,x);

(%o7)                                  log(x)
```

La fonction **depends** permet justement de déclarer des dépendances d'une fonction sur certaines variables afin d'en tenir compte dans des dérivations formelles. Si une dépendance n'a pas été déclarée, la dérivée sera considérée comme nulle. Voici un exemple pour illustrer ces dépendances :

```
(%i9) depends(u,t);

(%o9)                                  [u(t)]

(%i10) diff(F,t);

(%o10)                                    0

(%i11) depends(F,u);

(%o11)                                 [F(u)]

(%i12) diff(F,t);

                                       du dF
(%o12)                                 -- --
                                       dt du
```

La variable *dependencies* stocke les dépendances connues :

```
(%i13) dependencies;

(%o13)                        [a(x), u(t), F(u)]
```

Pour supprimer une dépendance, on utilise la fonction **remove** comme ceci :

```
(%i14) remove(F, dependency);

(%o14)                                 done

(%i15) dependencies;

(%o15)                           [a(x), u(t)]
```

8.2 Primitives

Les fonctions **antid** et **antidiff** fournissent les éléments permettant de transformer une expression par intégration par parties. Elles font partie du package *antid* qu'il faut charger explicitement :

```
load("antid")$
```

Si on représente l'intégration par partie symboliquement par l'identité : $\int uv' = uv - \int u'v$, la fonction **antid** appliquée à l'expression de la forme uv' renvoie une liste de deux éléments correspondant aux expressions uv et $-u'v$ respectivement. Par exemple :

```
(%i34) antid(x^n*%e^(y(x))*diff(y(x),x),x,y(x));

                            n   y(x)        n - 1   y(x)
(%o34)                    [x  %e    , - n x       %e    ]

(%i37) antid(2*u(x)*v(x)*diff(v(x),x),x,v(x));

                                 2         2     d
(%o37)                    [u(x) v (x), - v (x) (-- (u(x)))]
                                                dx
```

La fonction **antidiff** renvoie l'expression de l'intégration par parties :

```
(%i40) antidiff(2*u(x)*v(x)*diff(v(x),x),x,v(x));

                                      /
                                 2    [  2     d
(%o40)                    u(x) v (x) - I v (x) (-- (u(x))) dx
                                      ]         dx
                                      /
```

Cette expression utilise, dans sa représentation interne, le mot-clé *integrate* pour désigner l'intégrale.

9 Intégration

9.1 Calcul d'intégrales

Les fonctions présentées dans ce paragraphe tentent de déterminer des primitives formelles de fonctions. Pour obtenir des évaluations numériques d'intégrales simples, il faut utiliser les commandes du package *quadpack* présenté à la section 9.3.

• La fonction *integrate*

La principale fonction utilisée pour calculer des primitives ou des intégrales est **integrate**. La syntaxe peut prendre une des deux formes suivantes :

```
integrate(expr, x)
integrate(expr, x, a, b)
```

Les arguments a et b spécifient les bornes d'intégration. Par exemple :

```
(%i21) integrate(sin(x)^3,x);

                                   3
                              cos (x)
(%o21)                        ------- - cos(x)
                                 3

(%i22) integrate(sin(x)^3,x,0,%pi);

                                       4
(%o22)                                 -
                                       3
```

• La fonction *ldefint*

La fonction **ldefint** permet de calculer des intégrales avec des bornes indéfinies a et b en prenant la limite d'intégrales sur des intervalles définis et en faisant tendre les bornes vers a et b. Par exemple :

```
(%i5) assume(a>0);

(%o5)                                [a > 0]

(%i9) ldefint(exp(-a*x)*cos(x),x,0,inf);

                                       a
(%o9)                                 ------
                                       2
                                      a  + 1
```

• La fonction *changevar*

La fonction **changevar** effectue des changements de variables dans une intégrale. Sa syntaxe est :

```
changevar(expr, f(x,y), y, x)
```

Le changement de variable est exprimé sous la forme $f(y, x) = 0$ et la variable y remplace la variable x dans l'expression de départ. Par exemple :

```
(%i1) I: 'integrate(exp(-x^2)*x,x);

                                   /          2
                                   [       - x
(%o1)                              I x %e       dx
                                   ]
                                   /

(%i2) changevar(I,y=x^2,y,x);
```

```
                                  /
                                  [    - y
                                  I  %e    dy
                                  ]
                                  /
(%o2)                             ----------
                                       2
```

• La fonction *risch*

La fonction **risch** implémente l'algorithme de Risch qui tente de calculer, par des méthodes de calcul algébrique, des primitives pour des fonctions contenant des radicaux, des fractions rationnelles, des exponentielles et des logarithmes. Prenons par exemple la fonction suivante :

$$f(x) = \frac{-e^x - x + x\,\log x + x\,e^x\,\log x}{x\,\left(e^x + x\right)^2}$$

```
(%i15) f: (-%e^x-x+x*log(x)+x*%e^x*log(x))/(x*(%e^x+x)^2);

                          x                              x
                      x %e  log(x) + x log(x) - %e  - x
(%o15)                ---------------------------------
                                    x      2
                              x (%e  + x)
```

```
(%i16) risch(f,x);

                                  log(x)
(%o16)                          - -------
                                    x
                                  %e  + x
```

Une primitive de $f(x)$ est donc la fonction $F(x) = -\dfrac{\log x}{e^x + x}$.

9.2 Transformée de Laplace

La fonction **laplace** essaie de calculer la transformée de Laplace d'une fonction f :

$$F(s) = \int_0^\infty e^{-st}\,f(t)\,dt$$

Il faut préciser en arguments la variable t dont dépend l'expression et le nom de la variable de tranformation s. Par exemple :

```
(%i2) laplace(t^2,t,s);

                                  2
(%o2)                             --
                                   3
                                  s
```

```
(%i3) laplace(exp(-a*t),t,s);
```

```
                                        1
(%o3)                                 -----
                                      s + a
```

La transformée de Laplace inverse peut être calculée au moyen de la fonction **ilt**. Par exemple, en reprenant les deux exemples précédents :

```
(%i3) ilt(2/s^3,s,t);

                                        2
(%o3)                                   t

(%i4) ilt(1/(s+a),s,t);

                                      - a t
(%o4)                                %e
```

De manière générale, l'expression traitée par cette fonction doit être une fraction rationnelle dont le dénominateur ne comporte que des facteurs linéaires ou quadratiques.

Les deux fonctions **laplace** et **ilt** sont souvent utilisées conjointement avec **solve** et **linsolve** pour résoudre des équations différentielles ou de convolution.

9.3 Le package *quadpack*

Les fonctions de ce package, dérivé du package QUADPACK écrit en Fortran et faisant partie de la *SLATEC Common Mathematical Library*, permettent de calculer numériquement des intégrales en utilisant diverses techniques d'intégration numérique.

Huit fonctions sont disponibles et sont adaptées à divers types d'intégrales.

quad_qag
quad_qagi
quad_qagp
quad_qags
quad_qawc
quad_qawf
quad_qawo
quad_qaws

Leur syntaxe générale est la suivante :

```
quad_qag(f,x,a,b,key,[epsrel,epsabs,limit])
quad_qagi(f,x,a,b,[epsrel,epsabs,limit])
quad_qagp(f,x,a,b,points,[epsrel,epsabs,limit])
quad_qags(f,x,a,b,[epsrel,epsabs,limit])
quad_qawc(f,x,c,a,b,[epsrel,epsabs,limit])
quad_qawf(f,x,a,omega,trig,[epsabs,limit,maxp1,limlst])
quad_qawo(f,x,a,b,omega,trig,[epsrel,epsabs,limit,maxp1,limlst])
quad_qaws(f,x,a,b,alpha,beta,wfun,[epsrel,epsabs,limit])
```

La fonction **quad_qag**, par exemple, calcule des intégrales sur des intervalles finis en donnant le choix entre six types de formules de quadrature, désignées par des valeurs entières entre 1 et 6 qu'on spécifie dans l'argument *key*. Des valeurs plus élevées de cet argument conviennent pour des fonctions qui présentent de nombreuses oscillations. Par exemple, on peut calculer numériquement l'intégrale $\int_0^1 \sqrt{x}\log(1/x)\,dx$ comme ceci :

```
(%i42) quad_qag (x^(1/2)*log(1/x), x, 0, 1, 3, 'epsrel=5d-8);

(%o42)     [.4444444444574295, 8.737223572349164e-9, 899, 0]
```

Par comparaison, voici le résultat fourni pour la même intégrale par la fonction **integrate** :

```
(%i47) integrate(x^(1/2)*log(1/x), x, 0, 1);

                                       4
(%o47)                                 -
                                       9

(%i53) 4/9,float;

(%o53)                          .4444444444444444
```

Les arguments *epsrel*, *epsabs*, *limit*, etc. sont optionnels. Ils ont la signification suivante :
- *epsrel* est l'erreur relative souhaitée (par défaut 10^{-8}) ;
- *epsabs* est l'erreur absolue souhaitée (par défaut 0) ;
- *limit* est le nombre maximal d'itérations (par défaut 200).

La fonction renvoie une liste de quatre valeurs :
- la valeur approchée de l'intégrale ;
- l'erreur absolue estimée ;
- le nombre d'évaluations de l'intégrande ;
- un code d'erreur (0 si aucune erreur n'a été rencontrée).

10 Systèmes dynamiques

10.1 Équations de récurrence linéaires

La fonction principale permettant de résoudre des équations de récurrence linéaires s'appelle **solve_rec**.

Cette fonction fait partie du package de même nom qu'il faut charger explicitement au moyen de la fonction **batch** :

```
(%i1) batch(solve_rec)$
```

Voici un exemple de résolution d'une équation de récurrence linéaire d'ordre 1 à coefficients constants :

$$u_t - 4u_{t-1} = 3$$

```
(%i69) eqr: u[t]-4*u[t-1] = 3;

(%o69)                          u  - 4 u      = 3
                                 t      t - 1

(%i71) solve_rec(eqr, u[t]);

                                        t    t
(%o71)                        u  = %k  4  + 4  - 1
                               t      1
```

On peut passer en argument supplémentaire une valeur initiale. Par exemple, pour trouver la solution correspondant à $u_0 = 0$, on écrit :

```
(%i72) solve_rec(eqr, u[t], u[0] = 0);

                                          t
(%o72)                              u  = 4  - 1
                                     t
```

On utilise la même fonction pour résoudre des équations de récurrence d'ordre 2. Par exemple, prenons l'équation

$$6\,u_{t+2} - 5\,u_{t+1} + u_t = -2t - 3$$

On obtient :

```
(%i73) eqr: 6*u[t+2] - 5*u[t+1] + u[t] = -2*t-3;

(%o73)               6 u      - 5 u      + u  = - 2 t - 3
                          t + 2      t + 1    t

(%i74) solve_rec(eqr, u[t]);

                                    %k     %k
                                      1      2
(%o74)                         u  = --- + --- - t + 2
                                t    t      t
                                    3      2
```

Avec les conditions initiales $u_0 = 0$, $u_1 = -1$, on obtient :

```
(%i75) solve_rec(eqr, u[t], u[0]=0,u[1]=-1);

                                  3 - t      1 - t
(%o75)                    u  = - 2      + 2 3      - t + 2
                           t
```

10.2 Équations différentielles linéaires

La résolution des équations différentielles ordinaires se fait au moyen de la fonction **ode2**. Pour trouver la solution correspondant à des conditions données, on utilise l'une des fonctions **ic1**, **ic2** ou **bc2**. Les deux premières concernent des conditions initiales d'ordre 1 et 2 respectivement. La dernière concerne des conditions au bord à l'ordre 2.

Équation d'ordre 1

Prenons l'équation différentielle $y' - 2y = 7$:

```
(%i1) eqd: 'diff(y,t)-2*y = 7;

                                    dy
(%o1)                               -- - 2 y = 7
                                    dt

(%i2) sol: ode2(eqd,y,t);
```

```
                                          - 2 t
                                    7 %e          2 t
(%o2)                    y = (%c - ---------) %e
                                       2

(%i3) expand(sol);
                                          2 t   7
(%o3)                          y = %c %e    - -
                                            2
```

Pour trouver la solution correspondant à la condition initiale $y(0) = 5$, on utilise la fonction **ic1** comme ceci :

```
(%i4) ic1(sol, t=0, y=5);
                                          2 t
                                     17 %e    - 7
(%o4)                           y = ------------
                                          2
```

Équation d'ordre 2

Prenons l'équation différentielle $y'' + 3\,y' + 2\,y = t\,e^{-t}$:

```
(%i5) eqd: 'diff(y,t,2) + 3*'diff(y,t) + 2*y = t*%e^(-t);
                            2
                           d y     dy              - t
(%o5)                      --- + 3 -- + 2 y = t %e
                             2     dt
                           dt

(%i6) sol: ode2(eqd,y,t);
                     2              - t
                  (t  - 2 t + 2) %e             - t          - 2 t
(%o6)          y = -------------------- + %k1 %e    + %k2 %e
                            2
```

Pour trouver la solution correspondant aux conditions initiales $y(0) = -1$ et $y'(0) = -1$, on utilise la fonction **ic2** (abréviation de *initial conditions*) comme ceci :

```
(%i9) ic2(sol, t=0, y=-1, 'diff(y, t)=-1);
                       2              - t
                    (t  - 2 t + 2) %e           - t      - 2 t
(%o9)            y = -------------------- - 3 %e    + %e
                              2
```

Pour trouver la solution correspondant aux conditions initiales $y(0) = y(1) = 0$, on utilise la fonction **bc2** (abréviation de *boundary conditions*) comme ceci :

```
(%i14) bc2(sol, t=0, y=0, t=1, y=0);
                     2              - t              - t     1 - 2 t
                  (t  - 2 t + 2) %e       (%e - 2) %e      %e
(%o14)         y = -------------------- - -------------- - ---------
                            2                2 %e - 2      2 %e - 2
```

11 Entrées et sorties depuis des fichiers

Les fichiers d'instructions peuvent prendre trois formes différentes :

– fichiers constitués d'instructions Maxima. Ils peuvent être créés directement grâce à un éditeur de texte ou au moyen de la commande **stringout**. Ces fichiers de script ont une extension *.mac* ou *.mc* (ou l'une des extensions contenues dans la variable *file_type_maxima*) ;

– fichiers constitués d'instructions Lisp. Ils peuvent être créés par une commande **save** ou, à partir d'un fichier script Maxima, grâce à la commande **translate_file**. Ils ont en général une extension *.lisp* (ou l'une des extensions contenues dans la variable *file_type_lisp*) ;

– fichiers Lisp compilés. Ils sont chargés au moyen de la commande **load** et ont une extension *.fasl*. Ils peuvent être créés à partir d'un fichier script Maxima grâce à la commande **compile_file**.

11.1 Lecture de fichiers

Les fichiers Maxima sont recherchés sur tous les chemins d'accès contenus dans la variable *file_search_maxima*. Ils peuvent être lus et interpétés au moyen des commandes **batch** ou **batchload** :

– la commande **batch** lit les instructions du fichier script, les affiche, les exécute et affiche le résultat. Elle génère des étiquettes de type `%i` pour les instructions lues et de type `%o` pour leurs résultats ;

– la commande **batchload** lit et exécute les instructions sans les afficher et sans afficher le résultat (sauf s'il y a une commande **print** explicite). Elle ne génère pas d'étiquettes de type `%i` et `%o`.

On peut aussi lire des fichiers Maxima avec la commande **load**.

Les fichiers Lisp sont recherchés sur tous les chemins d'accès contenus dans la variable *file_search_lisp*. Ils peuvent être lus et interpétés au moyen des commandes **load** ou **loadfile** : la différence entre les deux est que la première recherche le chemin complet du fichier à exécuter en appelant la fonction **file_search** qui fouille tous les répertoires contenus dans les variables *file_search_maxima* et *file_search_lisp*. Le chemin complet est ensuite passé à la commande **loadfile**. Celle-ci peut être invoquée directement si le chemin du fichier est spécifié (de manière absolue ou relative au répertoire courant).

• La fonction *file_search*

La fonction **file_search** peut aussi être invoquée avec la syntaxe suivante :

```
file_search (filename, pathlist)
```

afin d'imposer de ne chercher que dans les répertoires spécifiés par le second argument.

La commande **load** peut lire des fichiers créés au moyen des fonctions **save**, **translate_file** et **compile_file** (qui génèrent du code Lisp), ou de la fonction **stringout** (qui crée du code Maxima). Selon qu'elle doit lire du code Lisp ou du code Maxima, elle appelle en fait les fonctions **loadfile** ou **batchload** respectivement. La fonction **file_type** indique de quel type de fichier il s'agit en examinant l'extension utilisée.

Chaque fois qu'une commande **load**, **loadfile** ou **batchload** est exécutée, le chemin complet du fichier est écrit dans la variable système *load_pathname* et cette variable peut être utilisée dans le courant du fichier lui-même. Cela ne marche pas avec la commande **batch**.

11.2 Écriture de fichiers

La commande **stringout** crée un fichier contenant des instructions Maxima tandis que la commande **save** crée un fichier contenant du code Lisp. La syntaxe de ces commandes peut prendre plusieurs formes :

```
save (filename, name_1, name_2, name_3, ...)
save (filename, values, functions, labels, ...)
save (filename, [m, n])
save (filename, name_1=expr_1, ...)
save (filename, all)
save (filename, name_1=expr_1, name_2=expr_2, ...)

stringout (filename, expr_1, expr_2, expr_3, ...)
stringout (filename, [m, n])
stringout (filename, input)
stringout (filename, functions)
stringout (filename, values)
```

Si on appelle ces fonctions avec un nom de fichier qui existe déjà, les nouvelles données peuvent remplacer les données existantes ou, au contraire, leur être ajoutées : cela dépend de la valeur de la variable globale *file_output_append*. Si cette variable est fixée à la valeur FALSE, les anciennes données sont écrasées par les nouvelles. Autrement les nouvelles données sont ajoutées à la suite des anciennes.

• La fonction *translate_file*

La fonction **translate_file** convertit un fichier d'instructions Maxima en un fichier d'instructions Lisp. La commande **compile_file** exécute la même conversion et compile le code obtenu. La syntaxe peut prendre plusieurs formes :

```
translate_file (maxima_filename)
translate_file (maxima_filename, lisp_filename)
compile_file (filename)
compile_file (filename, compiled_filename)
compile_file (filename, compiled_filename, lisp_filename)
```

• La fonction *with_stdout*

La fonction **with_stdout** écrit le résultat d'une ou de plusieurs instructions dans un fichier ou sur un canal de sortie. Le fichier texte ainsi créé peut être lu au moyen de la fonction **print_file**. Par exemple :

```
(%i21)  with_stdout ("~/tmp.out", for i:5 thru 10 do
      print (i, "! yields", i!))$
(%i22) printfile ("~/tmp.out")$

5 ! yields 120
6 ! yields 720
```

```
7 ! yields 5040
8 ! yields 40320
9 ! yields 362880
10 ! yields 3628800
```

• La fonction *writefile*

On peut enfin transcrire dans un fichier les instructions de la session courante en utilisant les fonctions **writefile** ou **appendfile** : la première écrase le contenu qui existe déjà éventuellement dans le fichier spécifié tandis que la seconde ajoute les nouvelles données aux données existantes. Les instructions sont transcrites à partir du moment où l'instruction est exécutée. Si le fichier n'existe pas encore, il est créé. Lorsqu'on veut arrêter la transcription, on referme un fichier au moyen de l'instruction **closefile**. Les commandes ainsi sauvegardées ne peuvent pas être chargées et réexécutées car elles sont simplement dans le format d'affichage de la console. Elles peuvent seulement être affichées grâce à la fonction **printfile**.

11.3 Chemins d'accès

Les fonctions **pathname_directory**, **pathname_name** et **pathname_type** permettent d'extraire les composantes d'un chemin d'accès. Elles renvoient respectivement le chemin du répertoire parent, le nom du fichier (sans l'extension) et l'extension du fichier. Par exemple :

```
(%i1) path: "/users/vhugo/maxima/myfile.mac";

(%o1)                    /users/vhugo/maxima/myfile.mac

(%i2) pathname_directory(path);

(%o2)                          /users/vhugo/maxima/

(%i3) pathname_name(path);

(%o3)                                   myfile

(%i4) pathname_type(path);

(%o4)                                     mac
```

• La fonction *filename_merge*

La fonction **filename_merge** permet de construire des chemins d'accès à partir de deux éléments. Par exemple :

```
(%i8) filename_merge("/users/vhugo/maxima/","quasimodo.mac");

(%o8)                    /users/vhugo/maxima/quasimodo.mac
```

11.4 Génération de commandes TEX

La fonction **tex** est la commande de plus haut niveau pour transformer une expression Maxima en une expression utilisable dans un fichier TEX (qui n'est pas forcément valide pour LATEX).

La fonction **tex1** produit une chaîne représentant l'expression mais sans la placer dans un environnement. Par exemple :

```
(%i11) e: sin(x)+cos(x);

(%o11)                          sin(x) + cos(x)

(%i12) tex(e);

(%o12)                               false

(%i13) tex1(e);

(%o13)                           \sin x+\cos x
```

Les fonctions **get_tex_environment** et **set_tex_environment** concernent les environnement TEX utilisés pour mettre en forme certains opérateurs qui apparaissent dans des expressions . Si aucun environnement n'est défini pour l'opérateur en question, un environnement est fourni par défaut par la fonction **get_tex_environment_default** : celui-ci peut être modifié au moyen de la fonction **set_tex_environment_default**.

• La fonction *texput*

La fonction **texput** permet de fabriquer de nouvelles constructions pour étendre les capacités de la fonction **tex**. Elle possède différentes syntaxes. En voici un exemple simple :

```
(%i14) texput (cov, "\\mathrm{cov}");

(%o14)                           \mathrm{cov}

(%i15) tex(cov(X,Y));

(%o15)                               false
```

Voici un autre exemple qui définit un préfixe :

```
(%i16) prefix ("grad");

(%o16)                               grad

(%i17) texput ("grad", " \\nabla ", prefix);

(%o17)                              \nabla

(%i18) tex (grad f);

(%o18)                               false

(%i19) tex1 (grad f);

(%o19)                              \nabla f
```

Génération de commandes LaTeX

Pour produire des commandes dans le format LaTeX au lieu du format Plain TeX, il faut charger le fichier *mactex-utilities*, comme ceci :

```
load("mactex-utilities")$
```

Ce package permet, d'une part, que les matrices soient codées avec l'environnement *pmatrix* plutôt qu'avec la macro du même nom (qui n'est pas reconnue par LaTeX) et, d'autre part, que les fractions utilisent la macro *frac* plutôt que la macro *over*. Voici quelques exemples qui montrent tout d'abord ce que génère par défaut la fonction **tex**, puis ce qu'on obtient avec la même fonction une fois que le package *mactex-utilities* a été chargé :

```
(%i1) m: matrix([1,2],[2,-1]);

                                    [ 1   2  ]
(%o1)                               [        ]
                                    [ 2  - 1 ]

(%i2) tex1(m);

(%o2)                      \pmatrix{1&2\cr 2&-1\cr }

(%i3) q: 2/3;

                                        2
(%o3)                                   -
                                        3

(%i4) tex1(q);

(%o4)                             {{2}\over{3}}

(%i5) load("mactex-utilities");

STYLE-WARNING: redefining MAXIMA::TEX-MATRIX in DEFUN
(%o5)  /opt/local/share/maxima/5.28.0/share/utils/mactex-utilities.lisp

(%i6) tex1(m);

(%o6)           \begin{pmatrix}1 & 2 \\ 2 & -1 \\ \end{pmatrix}

(%i7) tex1(q);

(%o7)                              \frac{2}{3}
```

12 Programmation avec Maxima

Maxima offre la possibilité de définir ses propres fonctions et donc d'étendre les fonctionnalités du programme. Il est donc aussi un langage de programmation.

12.1 Maxima et Lisp

Maxima est écrit en langage Lisp. Il est possible de communiquer entre les deux interpréteurs.

Il y a une convention de nommage qui permet de passer des variables et des fonctions d'un environnement de programmation à l'autre :

- un symbole Lisp qui commence avec un signe dollar (`$`) correspond à un symbole Maxima sans le dollar ;
- un symbole Maxima qui commence avec un point d'interrogation (`?`) correspond à un symbole Lisp sans le point d'interrogation.

Des symboles Lisp spéciaux tels que le tiret ou l'astérisque doivent être échappés quand ils apparaissent dans du code Maxima.

On peut exécuter du code Lisp dans une session Maxima en utilisant l'instruction spéciale `:lisp` comme par exemple :

```
:lisp (foo $x $y)
```

Cette instruction invoque la fonction Lisp *foo* avec les variables Maxima x et y comme arguments. Elle peut être utilisée dans des fichiers scripts exécutés par les commandes **batch** ou **demo** mais pas dans des fichiers lus au moyen des commandes **load**, **batchload**, **translate_file**, ou **compile_file**.

On peut basculer entre des sessions Lisp et Maxima depuis la console :

- la fonction **to_lisp** ouvre une session Lisp interactive :

```
to_lisp();
```

- on retourne à la session Maxima au moyen de la commande `(to-maxima)` qui referme la session Lisp.

```
(to-maxima)
```

Voici quelques exemples qui illustrent ces conventions :

```
(%i2) to_lisp();

Type (to-maxima) to restart, ($quit) to quit Maxima.

 (defun triple (x) (* 3 x))

TRIPLE

 (triple 6)

18

 (to-maxima)

(%o2)                                true

(%i3) :lisp (triple 8)
```

```
24

(%i3) ?triple(8);

(%o3)                                  24

(%i4) a: 5;

(%o4)                                   5

(%i5) to_lisp();

Type (to-maxima) to restart, ($quit) to quit Maxima.

 (triple $a)

15
```

12.2 Définition de fonctions

• La fonction *define*

La fonction **define** peut prendre plusieurs formes et est équivalente au symbole := servant à définir de nouvelles fonctions. La syntaxe est :

```
define (f(x_1, ..., x_n), expr)
define (f[x_1, ..., x_n], expr)
define (funmake (f, [x_1, ..., x_n]), expr)
define (arraymake (f, [x_1, ..., x_n]), expr)
define (ev (expr_1), expr_2)
```

Lorsque les arguments $x_1, \ldots, x_n$ sont placés entre parenthèses, on définit une fonction ordinaire ; lorsqu'ils sont placés entre crochets, on définit une fonction indexée.

Voici quelques exemples :

```
(%i72) define (F(x, y), cos(y) - sin(x));

(%o72)                    F(x, y) := cos(y) - sin(x)

(%i73) F(a,b);

(%o73)                           cos(b) - sin(a)

(%i74) define (G[m,n], m.n-n.m);

(%o74)                        G     := m . n - n . m
                               m, n

(%i75) G[a,b];

(%o75)                             a . b - b . a
```

Si le dernier ou l'unique argument d'une fonction est une liste à un élément, la fonction admet un nombre variable d'arguments. Par exemple :

```
(%i76) define (P([L]), '(apply ("*", L)));

(%o76)                          P([L]) := apply("*", L)

(%i77) P(r,s,t);

(%o77)                                    r s t
```

• La fonction *lambda*

Les expressions *lambda* sont des fonctions anonymes. Elles peuvent apparaître comme arguments d'autres fonctions qui attendent le nom d'une fonction. Par exemple :

```
(%i78) apply (lambda ([x], x^2), [a]);

                                             2
(%o78)                                      a
```

12.3 Le contrôle des flux

Maxima dispose de commandes conditionnelles qui permettent de contrôler les séquences d'instructions dans un fichier de script ou dans la définition d'une fonction.

12.3.1 Boucles

Les boucles itératives se font au moyen d'une commande **for** dont la syntaxe peut prendre plusieurs formes lorsqu'elle est couplée avec les mots-clés *thru*, *unless* et *while* :

for *variable* : *initial_value* step *increment* thru *limit* do *body*
for *variable* : *initial_value* step *increment* while *condition* do *body*
for *variable* : *initial_value* step *increment* unless *condition* do *body*

Les instructions définies par l'argument *body* sont exécutées jusqu'à ce que la variable de contrôle excède la limite imposée par le mot-clé *thru*, ou tant que la condition du mot-clé *unless* reste fausse ou que la condition du mot-clé *while* reste vraie. On peut spécifier plusieurs conditions de sortie : la boucle s'arrête dès que l'une d'entre elles est réalisée.

À l'intérieur du bloc attaché au mot-clé *do*, on peut insérer une commande **return** pour interrompre son exécution. On peut aussi utiliser une commande **go**.

La variable de contrôle est une variable locale à l'intérieur du bloc d'instructions. Elle ne modifie pas une variable extérieure qui aurait le même nom.

Une autre forme de la commande **for** consiste à parcourir les éléments d'une liste. La syntaxe est :

for *variable* in *list end_tests* do *body*

L'argument *end_tests* est optionnel et permet d'arrêter l'exécution du bloc d'instructions *body*.

12.3.2 Branchements

On fait des branchements conditionnels au moyen de la commande **if**. Celle-ci peut prendre plusieurs formes :

if *cond_1* then *expr_1* else *expr_0*
if *cond_1* then *expr_1* elseif *cond_2* then *expr_2* elseif ... else *expr_0*

Les arguments *cond* peuvent être n'importe quelle expression prenant les valeurs TRUE ou FALSE. Les symboles apparaissant dans ces conditions peuvent être les opérateurs de comparaison habituels : <, <=, =, >=, >. La négation de l'égalité est représentée par le symbole #. On peut aussi utiliser les mots-clés *equal*, *notequal*, *and*, *or* et *not*.

• La fonction *go*

La fonction **go** permet de faire des branchements inconditionnels. Elle dirige le flux d'exécution vers une instruction marquée par un tag spécifé. Le tag doit figurer dans le même bloc que la commande **go** : on ne peut pas faire un branchement vers un bloc externe.

Une série d'instructions utilisant cette commande ressemble à ceci :

```
block ([x], x:1, loop, x+1, ..., go(loop), ...)
```

Le tag, dans cet exemple, a été appelé *loop*.

• La fonction *return*

La fonction **return** permet d'interrompre l'exécution d'un bloc et de renvoyer une valeur.

12.3.3 Erreurs

• La fonction *error*

La fonction **error** a la syntaxe suivante :

```
error (expr_1, ..., expr_n)
```

Elle évalue et affiche les expressions passées en argument et provoque une erreur qui fait retourner au niveau supérieur.

La variable *error* est alors définie comme une liste qui reprend les diverses expressions et qui décrit l'erreur. Cette variable peut être affichée au moyen de la fonction **errormsg**.

• La fonction *errcatch*

La fonction **errcatch** permet au contraire de capturer les erreurs éventuelles et d'empêcher l'interruption de l'exécution des instructions. La syntaxe est :

```
errcatch (expr_1, ..., expr_n)
```

La fonction évalue les expressions passées en argument et renvoie normalement `[expr_n]`. Si une erreur se produit en cours d'évaluation, aucun autre argument n'est évalué : la fonction renvoie une liste vide et empêche l'erreur de se propager.

13 Optimisation

Le package *simplex* doit être chargé au moyen de la commande :

```
(%i1) load("simplex")$
```

Les fonctions **maximize_lp** et **minimize_lp** résolvent un problème d'optimisation linéaire par la méthode du simplexe. Le premier argument est l'expression de la fonction objectif et le second est une liste de contraintes (égalites ou inégalités).

Par exemple, considérons le programme suivant :

$$\begin{cases} \text{Max}\,(x_1 + 2x_2) \\ x_1 + 3x_2 \leq 21 \\ -x_1 + 3x_2 \leq 18 \\ x_1 - x_2 \leq 5 \\ x_1 \text{ et } x_2 \geq 0 \end{cases}$$

On le traite comme ceci :

```
(%i10) maximize_lp(x_1+3*x_2,[x_1+3*x_2<=21,-x_1+3*x_2 <=18,x_1-x_2<=5]);

                                      13        3
(%o10)                  [21, [x_2 = --, x_1 = -]]
                                      2         2
```

• La fonction *linear_program*

La fonction **linear_program** exécute l'algorithme du simplexe sur le système d'équations comportant les variables d'écart. Cette fonction est appelée par les deux fonctions précédentes. En reprenant l'exemple précédent, on écrit :

```
(%i15) A: matrix([1,3,1,0,0],[-1,3,0,1,0],[1,-1,0,0,1]);

                              [  1    3   1  0  0 ]
                              [                   ]
(%o15)                        [ - 1   3   0  1  0 ]
                              [                   ]
                              [  1   - 1  0  0  1 ]

(%i16) b: [21,18,5];

(%o16)                             [21, 18, 5]

(%i17) c: [-1,-2,0,0,0];

(%o17)                           [-1, -2, 0, 0, 0]

(%i18) linear_program(A, b, c);

(%o18)                       [[9, 4, 0, 15, 0], - 17]
```

Remarque : il faut changer les signes dans la fonction objectif.

14 Configuration

Au démarrage, Maxima recherche s'il existe, dans le répertoire courant de l'utilisateur, un répertoire appelé ".maxima" (avec un point initial). Ce répertoire peut contenir un fichier de démarrage appelé *maxima-init.mac* comportant des instructions Maxima à exécuter. On l'utilise comme fichier de configuration pour effectuer des réglages globaux affectant le comportement du programme.

On place typiquement dans ce fichier des instructions pour compléter la liste des chemins d'accès en ajoutant des répertoires supplémentaires ou pour redéfinir certains répertoires. Par exemple :

```
maxima_tempdir: "/tmp"
```

On peut aussi, dans le fichier de configuration, modifier le format d'affichage des nombres ou encore charger des packages comme par exemple :

```
load("mactex-utilities");
```

Index