Notes de cours programmation MAPLE

S. Tassart
Groupe Analyse-Synthèse
IRCAM, Institut de Recherche et de Coordination Acoustique–Musique
1 place Igor-Stravinsky, 75004 PARIS, FRANCE
tassart@ircam.fr

7 février 1997

Introduction

Ce document tente de présenter les principaux éléments d'information concernant la programmation sous MAPLE. Nous envisagerons la programmation sous différents aspects:

- l'utilisation et la création de nouvelles structures de données,
- l'utilisation de l'interface et la modification de l'interface utilisateur de MAPLE,
- l'utilisation conjointe de MAPLE avec d'autres environnements, et en particulier les possibilités d'entrées-sorties,
- la syntaxe des fonctions et des procédures, et le regroupement des fonctions au sein d'une bibliothèque.

Nous supposerons acquises les connaissances de base sur MAPLE, sa syntaxe, ses modes d'évaluation. Nous n'insisterons pas sur l'aspect mathématique de MAPLE si ce n'est au travers de quelques -trop rares- exemples.

Tous les exemples ont été testés sur les versions V-2 de MAPLE pour MS-Windows et pour stations de travail Unix. Les exemples encadrés sont les contenus de fichiers saisis à partir n'importe quel editeur de texte (Emacs ou Vi sous UNIX, TeachText sous MacIntosh, Bloc-Note ou Write sous Windows).

I Les structures de données

I.1 Les types

- Maple est un langage typé.
- Les types dans Maple tiennent beaucoup des langages orientés objet.

- Toutes les données sont typées hiérarchiquement. Par exemple:

```
type(5,nonnegint);   -> true
type(5,integer);     -> true
type(5,posint);      -> true
type(5,numeric);     -> true
type(5,constant);    -> true
```

- On obtient la liste de tous les types avec `?type`. On obtient une explication détaillée avec par exemple `?type,nonnegint`.
- Les types servent à vérifier que les arguments passés à une procédure sont du type attendu. Toutefois le typage n'est pas obligatoire. Par exemple, les deux procédures suivantes sont strictement équivalentes du point de vue de l'utilisateur:

```
fact1 := proc(n)
option remember;
if type(n,posint) then
   if n=1 then 1 else n*fact1(n-1) fi;
else ERROR(`Wrong type`);
fi;
end;

fact2 := proc(n:posint)
option remember;
   if n=1 then 1 else n*fact2(n-1) fi;
end;
```

- Pour vérifier le typage, 3 fonctions sont disponibles: `type(expr,_type)`, `whattype(expr)` et `hastype(expr,_type)`. `whattype()` renvoie une chaîne de caractère qui correspond au type *basique* auquel s'identifie l'expression. `type(expr,_type)` et `hastype(expr,_type)` sont des fonctions booléennes qui permettent de vérifier des types *complexes*.

```
g:= 3*x^2 + 2*x - 1;
type(g,polynom);             ->  true
whattype(g);                 ->  +
type(g,polynom(integer));    ->  true
```

- Créer un type *foo* est équivalent à créer une procédure `` `type/foo`(expr) `` qui renvoie `true` ou `false`. Malheureusement `whattype()` n'est pas redéfinissable.

  En fait, on distingue deux familles de types pour les utilisateurs:

  - les types *procéduraux*:

    ```
    `type/int100` := proc(f) evalb(type(f,nonnegint) and f<100) end;
    ```

  - les types *structurés*: (ou par *pattern matching*)

    ```
    `type/power` := anything^integer;
    ```

I.2 Les conversions

– Les conversions se font avec les fonctions suivantes:

```
`convert/f`(a,x,y,...) = convert(a,f,x,y,...)
```

– En algèbre linéaire, les conversions utiles sont celles qui concernent les *matrices*, les *tableaux*, les *vecteurs* et les *listes de listes* (`matrix`, `array`, `vector` et `listlist`) (cf. I.5.2)

```
> convert([a,b,c],`+`);
                    a+b+c
> convert([[a],[b],[c]],array);
                                [ a ]
                                [ b ]
                                [ c ]
```

I.3 Les simplifications implicites

il y a un certain nombre de simplifications automatiques dans Maple. Par exemple:

```
x and x -> x
x  +  0 -> x
x  -  x -> 0
1(x)    -> x
```

Les modifications implicites de Maple peuvent être gênantes. Par exemple, les deux expressions $a+b$ et $b+a$ sont équivalentes pour MAPLE dès lors qu'on utilise l'opérateur $+$. La commutativité implicite des opérateurs d'addition et de multiplication impose l'utilisation de nouveaux opérateurs `&+` et `&*` lorsque les additions ou les produits ne commutent pas. C'est pourquoi la multiplication matricielle se note `&*` et non `*`. C'est également la raison qui empêche de considérer des matrices blocs, car la multiplication des matrices utilise l'opérateur $*$ sur les éléments des matrices, qui ne peuvent donc pas être des sous-matrices blocs. L'exemple suivant donne un résultat différent en fonction du passé de l'interface (remember table):

```
> evalm([[ A , B ]] &* [[ X ], [ Y ]]);
        [ A X + B Y ]

> evalm([[ X ], [ Y ]] &* [[ A , B ]]);
        [ A X  X B ]
        [          ]
        [ Y A  B Y ]
```

I.4 Petit exemple

On veut implémenter la propriété suivante:

$$\forall n \in N \ \sin(n\pi) = 0 \tag{1}$$

On peut ecrire cette propriété de la façon suivante:

```
> oldsin := op(sin):
> sin := proc(x) if is(x/Pi,integer) then 0 else oldsin(x) fi; end:
```

Un exemple d'utilisation donnerait:

```
> sin(k*Pi);
            sin(k Pi)
> assume(k,integer): sin(k*Pi);
            0
```

I.5 List/Set, Table/Array/Vector/Matrix

I.5.1 List vs. Set

- []: une liste est une collection ordonnée d'éléments. La liste vide est `[]`.

```
c := [4,23,6];
a := [ op(c), op(c)];
```

- Une valeur `[[4,3],[2,8]]` est également du type `listlist`.
- {}: un ensemble est une collection qui ne retient ni l'ordre ni la multiplicité des éléments. L'ensemble vide est {}.

```
evalb({op(c),op(c)} = {op(c)})  -> true
```

- les ensembles sont manipulés par un certain nombre de fonctions telles que `union`, `intersect` ou `minus`.

I.5.2 Table vs. Array vs. Matrix vs. Vector

- Une table est un tableau associatif.

```
COLORS[bleu] := {bleu, blue, blau};
COLORS[rouge] := {rouge, red, rot};
print(COLORS)
   table([
        bleu  = {bleu, blue, blau},
        rouge = {rouge, red, rot}
         ]);
for c in entries(COLORS) do print(c) od;
```

- Un tableau est une table dont les index sont des entiers.
- Une matrice est un tableau avec 2 index.
- Un vecteur est un tableau avec 1 index.
- On peut convertir les matrices en listes de liste.

II L'interface

II.1 Des équivalences de fonctions

`type(a,f,x,y,...)`	$\Leftrightarrow$	`` `type/f`(a,x,y...) ``
`convert(a,f,x,y,...)`	$\Leftrightarrow$	`` `convert/f`(a,x,y...) ``
`simplify(a,f,x,y,...)`	$\Leftrightarrow$	`` `simplify/f`(a,x,y...) ``
`combine(a,f,x,y,...)`	$\Leftrightarrow$	`` `combine/f`(a,x,y...) ``
`define(a,f,x,y,...)`	$\Leftrightarrow$	`` `define/f`(a,x,y...) ``
`` `print/f`(x,y,...) ``	$\Leftrightarrow$	`f(x,y...)`
`evalf(a)`	$\leftrightarrow$	`` `evalf/kind_of_a`(a) ``
`?foo`	$\leftrightarrow$	``` `help/text/foo` := TEXT(``,...,``); ```
`?foo,x`	$\leftrightarrow$	``` `help/foo/text/x` := TEXT(``,...,``); ```
`readlib(pname)`	$\leftrightarrow$	``` read ``.libname.`/`.pname.`.m` ``` + remember table

`with(lname)`	$\leftrightarrow$	éventuellement lecture de `` `lname.m` ``
		+ affectations suivantes:
		`` lname[foo] := readlib(`lname/foo`); ``
		`foo := lname[foo];`
		(pour toutes les fonctions `foo` de la bibliothèque)

II.2 Les variables globales

Il y en a peu:

- `libname`: chemin pour les librairies,
- `constants`: liste des constantes déclarées,
- `Digits`: nombre de chiffres significatifs pour les calculs flottants[1],
- `Order`: ordre des développements limités,
- `printlevel`: pour le debug,
- `lasterror`: dernier message d'erreur,
- `status`: état interne de l'interprète.

II.3 L'interface

La fonction `interface` permet de modifier le comportement de l'interface. Nous donnons quelques exemples d'utilisation:

- `interface(verboseproc=2)`: pour afficher le source des fonctions,
- `interface(prettyprint=1)`: pour changer l'apparence du pretty-print,

1. `Digits` est limité à 100 sur la versin étudiante de MAPLE.

- `interface(prompt=‘Moi>‘)`: pour changer le prompt,
- `printlevel:=3`: niveau d'erreur affiché à l'écran,
- `aname()`: renvoie la liste des variables déjà utilisées (du type éventuellement passé en argument).

II.4 Fichier d'initialisation

Le système lit un fichier d'initialisation avant de commencer à être intercatif. C'est dans ce fichier d'initialisation que l'utilisateur peut initialiser les variables globales et pourquoi pas lire un certain nombre de packages automatiquement. Ce fichier est le fichier `MAPLE.INI` qui se trouve dans le répertoire courant sous DOS et `~/.mapleinit` pour Unix. Pour les autres systèmes, il faut consulter la documentation propre au système d'exploitation.

```
Order := 10: Digits := 20:
interface(prompt‘Moi > ‘):
interface(verboseproc=2):
```

II.5 Le debug

- `userinfo` et `infolevel`: dans une procédure, l'instruction `userinfo(5,grave,‘je suis ici‘)` n'affiche “je suis ici” que si `infolevel[grave] > 5`.
- `trace` et `untrace`: après que `trace(foo1,foo2)` a été évalué, tous les appels à `foo1` et `foo2` provoquent, en entrée l'affichage des arguments de la fonction, et en sortie un message indiquant que le déroulement de la fonction s'est achevé.
- `printlevel`: affiche tous les résultats intermédiaires correspondant à une profondeur inférieure à la valeur de `printlevel`. Le passage dans une structure de contrôle (`if` ou `for`) incrémente la profondeur de 1. Le passage dans une procédure incrémente la profondeur de 5.
- `profile`: permet d'obtenir des statistiques sur le fonctionnement d'une procédure dans un but d'optimisation.
- `mint`: utilitaire de vérification syntaxique.

II.6 Extension des commandes existantes

Le code de toutes les fonctions qui ne sont pas dans le noyau de MAPLE est accessible aux utilisateurs en utilisant `interface(verboseproc=2)` et `print`. De plus, les fonctions ne sont, en général, pas protégées. Il est donc tout à fait possible de récupérer le code existant pour une fonction de MAPLE, afin de la modifier et écraser l'ancienne fonction à l'aide de la fonction munie d'un nouveau code. Il s'agit en fait là de *remplacer* le code existant par un nouveau code. À partir de la version V-3, la commande `protect` empêche qu'un utilisateur écrase malencontreusement une fonction du système. Toutefois la fonction `unprotect` permet d'enlever une protection sur une fonction système, de remplacer malgré tout le code.

Il est également possible de rajouter de nouvelles caractéristiques à des fonctions de MAPLE. Nous avons vu que les fonctions `simplify`, `type` ou même `int` ne sont pas chacunes une unique fonction, mais une multitude de fonctions connues sous le nom de `‘symplify/foo‘` par exemple. En conséquence on peut rajouter une méthode afin que par exemple la fonction `simplify` sache simplifier un nouveau type de données.

En fait, MAPLE est un système (de ce point de vue) très ouvert, très modifiable et extensible. De nombreux rajouts, améliorations, packages sont disponibles dans le domaine public.

III Les entrées-sorties

MAPLE n'offre pas beaucoup de facilités pour communiquer avec le monde extérieur. En plus des fonctionnalités offertes par l'intermédiaire des menus déroulants qui dépendent fortement de la platteforme (MacIntosh, Windows, X-Window ...), il y a 6 instructions qui permettent de lire et d'écrire des fichiers: `save`, `read`, `writeto`, `appendto`, `readdata` et `readline`.

Les intructions `writeto` et `appendto` permettent de sauvegarder toutes les sorties du terminal sur un fichier. `writeto` écrase le fichier s'il existait déjà, `appendto` rajoute les lignes en fin du fichier. L'instruction `writeto(terminal)` (ou `appendto`) permet de revenir en mode de fonctionnement par défaut.

Les instructions `save` et `read` gerent la sauvegarde et la lecture de structures de données ou de programmes sur disque. Il existe deux formats de sauvegarde: le format interne MAPLE ou un format texte. Si le nom du fichier utilisé finit par l'extension `.m` alors la sauvegarde est en mode interne MAPLE[2], sinon elle est en mode texte. Ces instructions permettent de sauvegarder une ou plusieurs variables. Elles permettent également de sauvegarder l'ensemble de la session, c'est-à-dire l'état de l'interprète, toutes les variables d'interface, et toutes les variables, fonctions et procédures connues au moment de la sauvegarde.

Exemple: sauvegarde d'un plot au format MAPLE

```
myPlot := plot(sin(tan(x)),x=o..Pi);
save myPlot,‘result.m‘;
...
read ‘result.m‘;
```

Exemple: sauvegarde complète d'une session au format MAPLE

```
save ‘session.m‘;
```

L'instructions `readdata` permet de lire un fichier ASCII non formaté.

L'instruction `readline` associée à l'instruction `sscanf` permet de lire des fichiers ASCII formatée selon une syntaxe propre au fichier.

III.1 Les fichiers ASCII

III.1.1 Lecture

Les fichiers ASCII ne sont pas directement lisibles par MAPLE. En fait, pour qu'un fichier ASCII soit utilisable, il faut qu'il soit à la syntaxe MAPLE, c'est-à-dire qu'il soit une suite de

2. Les formats internes des versions V-2 et V-3 de MAPLE ont été changés. En conséquence, les sauvegardes en mode interne sur une version V-2 sont inutilisables sur des versions V-3 et réciproquement.

commandes et d'affectations séparées par des point-virgules.

- La façon la plus simple de lire un fichier de données consiste à utiliser la structure de liste, et à formater le fichier de la facon suivante:

`myData := [`	←	$1^{\text{ère}}$ ligne
10,		
24,		
-43,	←	des virgules
...		
65,		
98	←	pas de virgule
`];`	←	dernière ligne

ou bien, s'il n'est pas ennuyeux d'avoir un élément en trop:

`myData := [`	←	$1^{\text{ère}}$ ligne
10,		
24,		
-43,	←	des virgules
...		
65,		
98,	←	une virgule
`0];`	←	dernière ligne

- La fonction `readdata` est très utile pour charger des données numériques dans MAPLE. Le type des valeurs retournées par `readdata` est la liste de listes (`listlist`). On peut convertir ce type d'objet en vecteur, en matrice, ou en tableau. Supposons que le fichier de données `foo.dat` se présente sous la forme suivante:

12	4	54
78	32	22
89	-56	76
80	-4	-7
31	59	98

Alors, on peut obtenir un vecteur ou une matrice (et de façon plus générale un tableau) par les instructions suivantes:

```
> readlib(readdata);
> v1 := convert(readdata(`foo.dat`,integer,1),vector);
          v1 := [ 12, 78, 89, 80, 31 ]
```

```
> A  := convert(readdata(`foo.dat`,1),array);
                                 [ 12.    4.   54. ]
                                 [                 ]
                                 [ 78.   32.   22. ]
                                 [                 ]
                            A := [ 89.  -56.   76. ]
                                 [                 ]
                                 [ 80.   -4.   -7. ]
                                 [                 ]
                                 [ 31.   59.   98. ]
```

– La dernière méthode se rapproche beaucoup de la lecture des fichiers ASCII formatés en C. La fonction `readline(filename)` renvoie une chaîne de caractères correspondant à la ligne lue dans le fichier `filename`, et 0 en fin de fichier. Nous vous renvoyons à la documentation en ligne obtenue avec `? sscanf` et à la documentation de la fonction `fscanf` en C. Voici cependant un petit exemple d'utilisation qui consiste à faire la somme des deux premières colonnes du fichier `foo.dat`:

```
total1 := 0.0: total2 := 0.0:
line := readline(`foo.dat`):
while line <> 0 do
   temp := sscanf(",`%f %f`):
   total1 := total1 + temp[1]:
   total2 := total2 + temp[2]:
   line := readline(`foo.dat`):
od:
print(`TOTALS: `,total1,total2);   -> TOTALS: , 290., 35.
```

III.1.2 Écriture

Pour sauvegarder des fichiers ASCII, de deux choses l'une:

– pour l'utilisation ultérieure avec MAPLE, il suffit de sauvegarder en mode texte, le fichier est très facilement lisible (sauf s'il s'agit de sauvegarder des matrices!!!)

– pour l'utilisation avec d'autres programmes, il faut utiliser `writeto`, par exemple de la façon suivante:

```
printtofile := proc(filename, data)
  local f;
  writeto(filename);
  for f in data do lprint(f) od;
  writeto(terminal);
end:
```

IV Les procédures

- On obtient de l'aide avec `?proc`.
- Toute instruction MAPLE renvoie une valeur (parfois, la valeur est NULL)
- 3 syntaxes équivalentes sont disponibles pour décrire les expressions fonctionnelles ou procédurales.
  - La première est la définition standard la plus complète:

    ```
    proc(<param>)
      global <global>;
      local  <local>;
      option <options>.;
         <texte de la procedure>
    end;
    ```

  - La seconde correspond aux définitions fonctionnelles ou applicatives mathématiques.

    ```
    param -> local <local>; <texte de la procedure>;
    ```

    Avec la version V-3, il n'est plus possible de spécifier un champ *local* à l'intérieur d'une telle déclaration.
  - La dernière correspond à des définitions fonctionnelles ou procédurales très courtes.

    ```
    <<texte de la procedure>|<param>|<local>>
    ```

IV.1 Arguments

- Les arguments sont en général utilisés comme variables d'entrée.
- Les arguments sont évalués tous au moins une fois (sauf cas exceptionnels, c'est-à-dire des fonctions telles que `map`, `eval` ou `subs`).
- Les arguments ne sont pas forcément tous nommés:
  - `args`: séquence de tous les arguments passés à la procédure.
  - `nargs`: nombre d'arguments passés à la procédure.

  On obtient par exemple le i-ème argument avec `args[i]`.
- Les arguments ne sont pas forcement typés. À ce moment, il peut être utile de faire appel explicitement aux vérifications de type avec les fonctions `type()`.
- Les noms des arguments sont considérés comme des variables locales.
- Pour typer les arguments, on écrit:

```
foo := proc(arg1:type1, ...)
   ...
end;
```

– Les arguments peuvent être utilisés pour renvoyer des valeurs. Les fonctions classiques de Maple font rarement appel à cette caractéristique. Pour que l'affectation fonctionne, il faut passer en argument le nom de la variable et pas sa valeur, en conséquence il faut protéger le nom de la variable par des *simple quotes* "'".

```
r := irem(25,3,'q');
r;     -> 1
q;     -> 8
```

IV.2 Global

Le mot clef `global` est suivi de la séquence des variables déclarées comme globales. Ces déclarations sont optionnelles en V-2 car toute variable non déclarée est globale. Par contre en V-3, toute variable non déclarée est *susceptible*, sous certaines conditions, d'être locale. En conséquence, à partir de la version V-3, il faut décrire proprement quelles sont les variables locales et quelles sont les variables globales.

IV.3 Local

Le mot clef `local` est suivi de la séquence des variables locales qui ne sont visibles qu'à l'intérieur de la procédure.

– **DANGER: les procédures imbriquées ne partagent pas leurs variables locales.**

IV.4 Option

Deux syntaxes sont acceptées: `option` ou `options` (c'est-à-dire avec ou sans le "s"). Parmi toutes les options qui sont reconnues par MAPLE (`remember`, `operator`, `builtin`, `system`, `trace`, `angle`, `arrow`), seule la première est importante.

– `remember`: Cette option permet à la fonction de réserver de la mémoire pour une table de "souvenirs" de la fonction. Cela permet de garder en mémoire les résultats de tous les appels à cette fonction, et évite de réexécuter le code de la fonction quand le résultat est déjà disponible. Les meilleurs exemples consistent en des définitions de suites récursives.

  La "remember table" reste en mémoire tant que la fonction existe. Seules les fonctions définies avec l'option `system` pour lesquelles la table est "nettoyée" périodiquement (pendant les période dites de "garbage collecting") échappent à cette règle. En ce qui concerne les fonctions classiques, l'instruction `forget()` permet de forcer une fonction à oublier son passé.

  Il est dangeureux d'utiliser cette option dans le cas des fonctions qui ont des effets de bords (c'est-à-dire qui modifient l'interface, les variables globales, ou qui font de l'affichage), car dans le cas d'un appel de la fonction avec des arguments qui correspondent déjà à un résultat connu, le code n'est pas évalué, et les effets de bord ne se produisent pas.

– `operator`: sont considérées comme `operator` toutes les procédures qui sont définies avec la syntaxe `->` (arrow) ou la syntaxe `<>` (angle).

IV.5 Valeur de retour d'une procédure

Par défaut, une procédure renvoie la valeur de la dernière expression évaluée. Par exemple, la procédure `x -> x^2` renvoie la dernière expression évalué, c'est-à-dire le carré de l'argument.

```
mymax := proc(a,b) if a>b then a else b fi: end;
```

Le déroulement du code d'une procédure est interrompu par l'instruction `RETURN()`. La valeur de retour est alors la valeur passée en argument de la fonction `RETURN()`.

IV.6 Gestion des erreurs

La fonction `ERROR()` déclenche une erreur, interrompt le déroulement d'une procédure, et renvoie un message d'erreur système accompagné de la chaîne de caractères passée en argument de `ERROR()`.

La fonction `traperror()` permet de vérifier que l'instruction passée en argument se déroule avec ou sans erreur, sans pour autant interrompre le déroulement de la procédure.

IV.7 Composition

Les opérateurs `@` et `@@` permettent de composer des procédures ou fonctions entre elles.

- `f@g` correspond à la composition de f par g, ($(f@g)(2) = f(g(2))$).
- `f@@n` correspond à la composition itérée n fois de f, ($(f@@4)(2) = f(f(f(f(2))))$).

Par exemple on trouve:

```
f:=x->x+1: (f@@6)(x);                ---> x+6
h:=<x+1|x>: (h@@6)(x);               ---> x+6
g:=proc(x) x+1 end: (g@@6)(x);       ---> x+6
```

IV.8 Exemple

Voyons par exemple comment nous pourrions définir la fonction *max* définie par les propriétés suivantes:

$$\begin{aligned}
\max(a,b) &= \max(b,a) \\
\max(\max(a,b),c) &= \max(a,b,c) \\
\max(+\infty,a) &= +\infty \\
\max(-\infty,a) &= a \\
\max(a,a) &= \max(a) \\
\max(a) &= a \\
\max() &= -\infty
\end{aligned}$$

```
max := proc() local a;
  a := {args};
  if member(infinity,a) then RETURN(infinity) fi;
  a := map(
```

```
    proc(x) if type(x,function) and op(0,x) = max then op(x) else x fi end,
    a);
  a := map (
    proc(x) if x<> -infinity then x fi end,
    a);
  if nops(a) = 0 then RETURN(-infinity) fi;
  if nops(a) = 1 then RETURN(op(1,a)) fi;
  'max'(op(a))
end;
```

La première commande `a:={args};` permet d'implémenter la commutativité et d'éliminer les termes de multiplicité supérieure à un. La seconde implémente la propriété $\max(+\infty, a) = +\infty$. La troisième commande implémente la propriété $\max(\max(a, b), c) = \max(a, b, c)$ en évaluant toutes les occurances de `max` a ses propres arguments. La quatrième commande supprime toutes les occurances de $-\infty$ dans la liste des arguments. Les deux commandes suivantes évaluent `max` dans le cas où il ne reste plus que un ou aucun argument. La dernière commande renvoie `'max'(op(a))` sans l'évaluer.

Le comportement de la fonction `max` correspond dans un premier temps à une série de simplifications de l'expression passée en argument, et le cas échéant au retour de l'expression originale sans vraiment l'évaluer. C'est un comportement très standart en programmation MAPLE, (et très différents des autres styles de programmation). La plupart des fonctions de MAPLE (`sin`, `exp`, ...) se contentent de faire des manipulations de l'expression passée en argument quand c'est possible, et renvoie l'expression originale sans évaluation par défaut. Dans le cas de `sin`, le système vérifie si l'argument est une expression connue en π (si multiple entier de π alors 0, si $\pi/6$ alors $\frac{1}{2}$, etc), puis tente de ramener l'expression dans l'intervalle $]-\pi \ldots \pi]$, puis si l'expression est négative, sort le $-$ du sinus... Si aucunes de ces manipulations n'aboutissent, le système renvoie `'sin'(op(a))`, c'est à dire qu'il dit: $\sin(x) = \sin(x)$. En fait, c'est uniquement au moment de l'évaluation numérique (dans la fonction `` `evalf/sin`() ``) que *sin* fait réellement quelque chose, c'est-à-dire renvoie une valeur numérique.

Pour résumer, les propriétés intrinsèques d'une fonction f sont décrites dans la définition de la procédure MAPLE `f`. Les propriétés de f vis à vis des autres fonctions sont décrites dans la famille de fonctions `simplify` et `convert`. Les évaluations numériques sont assurées par la fonction `evalf/f`. La dérivée de la fonction se retrouve dans `diff/f`; la primitive . Je ne sais pas encore comment sont codés les primitives et les methodes d'intégration pour la fonction f.

V Les packages

- Il y a plusieurs types de fonctions et procédures:
  - les "builtins": construites dans le noyau. Leur source n'est pas accessible. Programmées et optimisées en C. Ce sont des fonctions qui peuvent ne pas être programmées autrement qu'en C (elles accèdent à des champs "privés" des structures internes de MAPLE)
  - les "autoloadables": elles ne sont en mémoire que lorsqu'elles ont été appelées au moins une fois. L'appel est automatique, et se fait en fait par l'astuce suivante:

```
foo := readlib(`foo`);
```

  - les “loadables”: elles ne sont pas directement accessibles. L’utilisateur doit taper explicitement `readlib(‘foo‘);` pour que la fonction soit visible à partir de l’interface.
  - les “packages”: les fonctions de packages ne sont appelables que par deux méthodes:
    - `lname[foo]();`
    - `with(lname); foo();`

    Une fois qu’elles sont en mémoire, elles y restent.

- Il y a beaucoup de packages qui s’appliquent à des champs d’activité très différents les uns des autres (`student`, `linalg`, `plots` ...).

```
with()      (-> linalg)
readlib()   (-> C() ou Fortran() )
lib[name]   (-> linalg)
```

V.1 Linalg

- Il y a plus de 100 fonctions disponibles dans ce *package*.
- Pour faire du calcul matriciel, calculer des déterminants, calculer des noyaux, exprimer sous la forme d’une suite récurrente, l’expression d’un déterminant ...
- Maple ne gère pas les matrices de taille non définie, pas plus que les matrices définies par bloc.
- Les matrices sont créées soit en exprimant la valeur de chacun de ses éléments, soit en exprimant la valeur de certains de ces éléments et des propriétés générant tous les autres éléments. Les matrices *creuses* sont des matrices pour lesquelles tous les éléments non définis sont nuls. Les matrices *bandes* sont entièrement définies par les valeurs de leurs diagonales. La matrice identité est définie soit par la propriété d’identité, soit par `&*()`. Il y a plein d’autres matrices disponibles.

```
> M := array(1..2,1..6,[(1,3)=p],sparse);
                  [ 0  0  p  0  0  0 ]
             M := [                  ]
                  [ 0  0  0  0  0  0 ]
> M[1,5] := p^2;
                                              2
                           M[1, 5] := p

> evalm(M);
                      [                2    ]
                      [ 0  0  p  0  p     0 ]
                      [                     ]
                      [ 0  0  0  0    0   0 ]
```

- Les opérations matricielles élémentaire sont: `+`, `&*`, `^(-1)`, `^(n)`, `*` et `&*()` pour respectivement l'addition matricielle, la multiplication matricielle, l'inversion matricielle, l'élévation à la puissance (en $\ln_2(n)$), la multiplication par un scalaire, et la matrice identité.
- exemple: calcul de déterminant, d'inverse, de résolution de systèmes linéaires (voir `?linalg linalg,det`, `linalg,inverse`, `linalg,linsolve`).

V.2 Création d'un package

- L'utilitaire `march` permet de fabriquer une archive de procédures, de rajouter des procédures à une archive. C'est l'équivalent de `ar` en C. Ceci dit, je trouve que la fonction est peu utile.
- Supposons que l'on veuille créer un package `foo`. Voilà l'approche que je préconise. Elle n'est pas unique, voire pas optimale, mais elle a le mérite de marcher tout à fait correctement.

```
[...]
read foo_hlp;     # lecture du fichier contenant l'aide in-line
[...]

# declaration des variables globales au package
`foo/globalx` := [...];
[...]

# code du package
`foo/x` := proc()
 [...]
 end;

 [...]

# fin du code

# Declaration des fonctions a 'exporter' pour l'utilisateur
foo[x] := readlib(`foo/x`);
[...]

# Sauvegarde sous forme interne du package
save `foo.m`;
```

- Le fichier `foo.m` doit être installé dans un des chemins indiqués par `libname`. La bibliothèque se lit avec l'instruction `with(foo);`.
- L'utilitaire `helptomaple` permet de générer automatique un fichier d'aide. Il transforme un fichier texte en fichier ASCII utilisable par MAPLE. Toutefois, il n'est pas possible de modifier le comportement du “browser”.

- Il n'est pas indispensable d'ouvrir une session interactive MAPLE pour faire du calcul formel. L'instruction `maple -q < myProg.mat > maple.log` permet d'interpréter toutes les commandes du fichier `myProg.mat` et de noter tous les résultats sur le fichier `maple.log`. Les noms des extensions ne sont pas significatifs.

  L'utilisation non-interactive de MAPLE et de `helptomaple` permet de gérer des projets avec des *Makefile*.

- Il vaut mieux éviter d'utiliser `with()` à l'intérieur du source du package et n'appeler les fonctions qui proviennent d'autres packages qu'avec `linalg[lname](foo);`.

V.3 Application

Création d'une bibliothèque qui utilise un type fabriqué dans la bibliothèque pour gérer les systemes linéaires en variables d'état en automatique linéaire.

Index

Table des matières