UPS – Centre d'Orsay
Mathématiques

MAO – Maîtrise
Année 2003–2004

MAPLE/ 0

Introduction à MAPLE

> Wir waren kompliziert genug, die Maschine zu bauen, und wir sind zu primitiv, uns von ihr bedienen zu lassen.
>
> Karl Kraus, *Untergang der Welt durch schwartze Magie*, 1922.
>
> Any sufficiently advanced technology is indistinguishable from magic.
>
> Artur C. Clarke, « *Technology and the Future* ». *Report on Planet Three*, 1972.

1. INTRODUCTION

Le but de ce module est d'apprendre à utiliser un système de calcul formel pour

(1) consolider des connaissances acquises en les voyant sous un jour pratique.
(2) expérimenter pour « découvrir » des conjectures et les tester, avant de tenter de les démontrer.

On s'initiera au passage aux méthodes du calcul formel et à l'analyse de leurs performances. Les thèmes étudiés seront essentiellement arithmétiques. C'est en partie un cours d'algorithmique, puisqu'on ne s'interdira pas de comparer diverses solutions, ni d'en rechercher qui soient efficaces. Ce n'est pas un cours de programmation.

Nous utiliserons le système MAPLE [1], qui est un logiciel de calcul formel puissant et d'utilisation simple, créé en 1981 par l'université de Waterloo (Canada). Il est recommandé de connaître l'anglais pour s'en servir, en particulier pour utiliser l'aide en ligne. Ce système n'est pas parfait : il est assez lent et il arrive fréquemment qu'un bug fasse perdre une session (sauver fréquemment !) ; il est inadapté au calcul numérique ou à la réalisation de grands projets. Mais il offre un langage relativement intuitif permettant de manipuler facilement les objets mathématiques courants, d'écrire et de tester rapidement de petits programmes.

Nous allons survoler divers aspects de MAPLE dans les pages suivantes. Le but étant de poser (et résoudre) des problèmes mathématiques le plus vite possible, on sera bref. Pour chaque nouvelle commande ou objet introduit au cours des TP, il est conseillé de

(1) lire l'aide en ligne.
(2) l'essayer sur des exemples, y compris pathologiques, en essayant de prévoir les résultats et de comprendre les réactions du système.

Il existe des heures de libre-service. Il faudra en profiter pour se familiariser aussi vite que possible avec MAPLE et sa syntaxe, ainsi qu'avec les notions introduites en TP : MAO = Mathématiques *Assistées* (pas Asservies ou Aseptisées) par l'Ordinateur.

RÉFÉRENCES

[1] J.-M. CORNIL & P. TESTUT, *Maple, introduction raisonnée*, Springer, 1995.
[2] B. PERRIN-RIOU, *Maple : Arithmétique et Algèbre*, Université Paris-Sud, 1998.
[3] D. REDFERN, *The Maple Handbook (Maple V.4)*, Springer, 1996.
[4] M. RYBOWICZ & J.-P. MASSIAS, *Maple V release 4*, Eyrolles, 1996.

[1] pour 'érable' ou MA*thematical* PLE*asure*.

2. Prise en main

Ces notes sont valables pour Maple V et plus, interface graphique `xmaple`.

Prompt : l'invite `>` (prompt) indique que Maple attend une instruction.

Exécuter : chaque instruction se termine par `;` (résultat affiché) ou `:` (résultat non affiché). Un retour chariot (`Enter`) déclenche l'exécution du groupe de commande en cours, *i.e.* le texte délimité par un [à gauche. Appuyer simultanément sur `Shift` et `Enter` passe à la ligne *sans* valider la commande : pratique pour écrire lisiblement. Il est conseillé d'utiliser un maximum de `:` pour cacher les résultats intermédiaires.

Charger un fichier : on peut lire/sauver un fichier au format `mws` (Maple WorkSheet) à partir du menu `File`. Si vous avez l'habitude d'un bon éditeur de textes (Emacs, Vim), il est préférable d'écrire tout programme un peu long sous forme de fichier *texte*, puis de les charger dans la session via `read` : le comportement est le même que si les commandes avaient été rentrées au clavier.

Aide en ligne : `?`*commande*

Redémarrer : `restart ;` (réinitialise tous les objets : variables globales, fonctions).

Commentaire : `#`*blabla* (le reste de la ligne est ignoré)

Dernière expression évaluée : `%` (l'avant dernière est `%%`, etc.).

3. Constantes et opérations élémentaires

Une suite de chiffres comportant un `.` (remplace la virgule française) ou un `e` (notation scientifique) est un réel. Si l'expression contient `I` (pas `i` !), c'est un complexe. Voir `type` et `whattype`.

Les opérations arithmétiques élémentaires sont notées `+`, `-`, `*`, `/`, `^` (exponentiation). Les fonctions mathématiques usuelles (exp/log, trigonométrie) s'obtiennent de façon évidente. Sauf valeurs particulières, si l'argument a des composantes exactes, le résultat sera formel, parfois identique à la formule tapée [*comparer* `sin(1)`, `sin(1.)`, `sin(Pi/3)`, `sin(Pi*1.3)`]. La commande `eval` et ses variantes, en particulier `evalf` pour une évaluation numérique, forcent Maple à évaluer plus on moins complètement une expression. Essayer

```
> Pi ; evalf(Pi) ; evalf(Pi, 100) ;
```

On peut modifier la variable globale `Digits` [*par exemple,* `Digits := 50`], pour fixer la précision courante des calculs à venir. En général, préférer `evalf`.

Exemples : À quoi correspond `pi` ? Obtenir une valeur approchée de $\sin(\pi/13)$. Que se passe-t'il pour $\sin(\pi/12)$?

4. Variables

Affectation : `x := 1` pour 'x reçoit 1'. La valeur à droite du signe `:=` est associée au nom (identificateur) situé à gauche.

Quelques pièges :

• Ne pas oublier le ':', sinon (`x = 1`) c'est un test d'égalité.

• Une variable libre x (non affectée) est un symbole : sa valeur est un "pointeur sur elle-même" (sera remplacée par le contenu de x quand on lui en donnera un). Essayer :

```
> restart ;
> P := x^2 ; x := 2 ;
```

```
> P ; Q := x - 1 ;
```

Sauf exception[2], un résultat est complètement évalué : toutes les variables sont remplacées par leurs valeurs. Utiliser `eval(P, 1)` pour une évaluation au premier rang au lieu d'une évaluation complète (permet de déterminer que `P = x^2`, même après l'affectation de `x`). `eval(A)` force au contraire une évaluation totale (par exemple pour le tableau de la note).

Empêcher l'évaluation :

Une expression entre *guillemets simples* (accent aigu) retarde l'évaluation : `'x'` est évalué en la variable formelle `x`, même si on a déjà donné une valeur à `x`. [*Expliquer l'effet de* `x := 1 ; x := 'x' ;`. *Vérifier avec* `whattype`]. Prévoir le résultat de

```
> restart ; x := y ; y := x ; x^2 ;
> restart ; x := y ; y := 'x' ; x^2 ;
```

De façon générale, pour éviter les problèmes : ne pas utiliser les mêmes identificateurs pour les variables libres (polynômes, etc.) et les autres (paramètres, compteurs, etc.) ; pour évaluer une expressions en un point, utiliser `subs` plutôt qu'une affectation.

Une expression entre "*guillemets graves*" (`‘x‘`) est un nom (chaîne de caractères). Exemple : `print(‘Bonjour‘) ;`. Pour concaténer objets ou noms, on peut utiliser '`.`' :

```
> x := 2 ;
> print(‘la valeur de x est ‘ . x)
```

Pour des expressions plus complexes il est plus agréable d'utiliser `printf`.

5. Quelques objets utiles

Intervalles : `1..100`

Séquences (*sequence*) **:** suite (ordonnée) d'objets séparés par des virgules. La séquence vide est appelée `NULL` (utile pour initialiser), qui n'est pas affichée par Maple. On peut concaténer deux séquences en les séparant par une virgule : `s1, s2`.

Un constructeur utile : `seq(f(i), i = 1..100)`. L'opérateur `$` a une fonction analogue, en moins lisible : `f(i) $i=1..100`, mais il peut aussi servir a initialiser une liste d'objets identiques `1 $ 100` ou un intervalle d'entiers `$ 1..100`. `seq` protège ses arguments contre une évaluation prématurée, `$` non. En conséquence son emploi est plus délicat, donc non recommandé ; comparer

```
> i :=10 ;
> seq(a[i], i = 1..10) ;
> a[i] $ i = 1..10 ;
> a[i] $ 'i' = 1..10 ;
> 'a[i]' $ 'i' = 1..10 ;
```

Listes (*list*) **:** de la forme `[` *sequence* `]`. C'est une séquence vue comme un seul objet, et non comme un agrégat. Réfléchir à la différence entre `f(`*liste*`)` et `f(`*séquence*`)`.

Ensembles (*set*) **:** de la forme { *sequence* }. C'est une liste non ordonnée (en particulier les éléments en double sont éliminés). Attention : l'ordre des éléments d'un ensemble est arbitraire et peut changer d'une session sur l'autre.

Tableau (*array*) **:** à une ou plusieurs dimensions. Par exemple, `A := array(1..20, 1..10)` est une matrice 20×10. Plus général qu'une liste, et traité plus efficacement par

[2] s'il n'est pas de type tableau ou dérivé (matrices, etc.) ou fonction, auquel cas seul son nom apparaîtrait. Essayer `A :=array([1,2,3]) ; A ;`

le système. Il n'est pas facile de déterminer automatiquement les dimensions d'un tableau [`op(2, eval(A))` *renvoie la séquence d'intervalles qui les constitue*]. On convertit un tableau en liste via `convert(A, list)` ; l'opération inverse est `convert(L, array)` ou plus simplement `array(L)`.

Tableau associatif (*hash table*) : plus général qu'un tableau : à des indices de type quelconque on associe une valeur. Par exemple, si on assigne `A[1] := 0` sans définir `A` au préalable, ce dernier est de type table. La *remember table* d'une fonction est une table (voir plus bas).

Pour extraire le i-ème élément d'une séquence/liste/ensemble/tableau : `L[i]`, $i \geqslant 1$. On peut changer les valeurs d'une liste/tableau : `L[i] := 1`. Si une liste a plus de 100 éléments MAPLE refuse [*essayer*] : utiliser un `array` ou lieu d'une liste.

Le nombre d'éléments d'une séquence/liste/ensemble est donné par `nops(L)`, et les éléments eux-mêmes, c'est-à-dire la séquence sous-jacente, par `op(L)`. On peut utiliser `op(i, L)` au lieu de `L[i]`. On peut extraire des intervalles : `op(3..10, L)`. De façon générale, on peut compter ou extraire les sous-objets de n'importe quelle expression. Par exemple

• `a := (x^2 + 4*y^2 = (x+1)*(5*x-2)) ;` [*analyser les composants de* `a` *(et de ses sous-composants) avec* `op`*. Extraire le 5 à l'aide d'une suite de* `op`]

• Le quatrième "élément" d'une fonction contient la table des valeurs spéciales (*remember table*). Essayer `op(4, eval(sin))`. Pour la plupart des fonctions, MAPLE va remplir cette table de lui-même à chaque fois qu'un nouveau résultat est calculé. Il est possible de remplir cette table soi-même ! [*essayer* `sin(1):=0 ;` (!!!)].

Si vous voulez savoir comment sont définies les fonctions MAPLE, vous pouvez les « afficher » : `interface(verboseproc = 3) ; print(sin) ;`

6. FONCTIONS

On peut évaluer une expression contenant des variables libres avec `subs`.

```
> f := x + log(y) + exp(z) ;
> subs({x=1, y=2}, f) ;
```

Mais `f` ne définit pas une fonction au sens habituel du terme [*essayer* `f(1)`]. Pour ce faire, on utilise la notation `->` : `F := (x, y) -> x + log(y) + exp(z) ;` associe au nom `F` une fonction de deux variables (dont la valeur est une expression formelle en `z`) [*que se passe-t-il si on oublie les parenthèse autour de* `x,y` *?*]. Un raccourci si l'expression existe déjà : `F := unapply(f, x,y) ;`

Dans l'autre sens, si `x` et `y` sont libres, `F(x, y)` est une expression formelle égale à `f`.

Procédures : pour des fonctions plus compliquées, utiliser `proc` :

```
> f := proc(séquence d'arguments)
    local séquence de variable locales ;
    global séquence de variable globales ;
    options séquence d'options ;
    ... ;
  end ;
```

`local`, `global` et `options` sont facultatives. La valeur de retour est la dernière expression évaluée. Utiliser `RETURN(`x`)` pour quitter la procédure prématurément. Il est interdit de modifier un argument d'une fonction dans le corps de celle-ci :

```
> f := proc(x) ; ... x := x+1 ; ... ; end ;
```

```
Illegal use of a formal parameter.
```

Remarque : les cas particuliers se codent facilement grâce à la *remember table*. Au lieu de

```
f := proc(x)
  if (x = 0)  then
    RETURN (1) ;
  fi ;
  sin(x) / x ;
end ;
```

on peut écrire

```
f := proc(x)
  sin(x) / x ;
end ;
f(0) := 1 ;
```

ou plus simplement (`proc` ne se justifie pas ici)

```
f := x -> sin(x) / x ;
f(0) := 1 ;
```

[*Que se passe-t-il si on intervertit ces deux lignes ?*].
Si `options` contient `remember`, la *remember table* se remplit toute seule à chaque appel de la fonction (utile pour certaines fonctions récursives, mais gourmand en mémoire).

Graphe : pour un graphe simple : `plot(f(x), x=-1..1)`. Comme `plot` ne protège pas ses arguments, il peut être nécessaire d'empêcher l'évaluation prématurée de la fonction : expliquer

```
> pi := x -> nops(select(isprime, [$1..x]))
> plot(pi(x), x=1..100) ;
> plot('pi(x)', x=1..100) ;
```

7. Programmation

Branchement conditionnel :

```
if test_1 then        # si ... alors
  ...
elif test_2 then      # sinon si ... alors
  ...
elif test_n then
  ...
else                  # sinon
  ...
fi ;
```

Un *test* est une expression à valeur "booléenne" (`true`, `false` ou `FAIL`). Seule la valeur `true` enclenche le branchement. Le `else` et les `elif` (= else if) sont optionnels. On peut utiliser les opérateurs logique `and`, `or`, ou `not` pour les tests. Exemple de tests :

- `x <= 0` (pour $\leqslant$)
- `x <> y` ou bien `not(x = y)` (pour $x \neq y$)
- `(type(p, integer) and isprime(p))`.

Boucles :
Avec compteur numérique

```
for i from min to max by pas    # i = min, min + pas, ...
do
  ...
od;
```

`from` et `by` sont optionnels (*min* et *pas* valent 1 par défaut). *min*, *max*, *pas* ne sont pas nécessairement entiers, ni *max* égale à la valeur finale de l'indice[3]. Si $min > max$, la boucle n'est pas exécutée. Attention, MAPLE n'est pas capable d'évaluer tout seul une borne formelle, il faut parfois l'aider avec `evalf` :

```
> for i to exp(1) do print(i) od;
Error, final value in for loop must be numeric or character
> for i to evalf(exp(1)) do print(i) od;
                        1
                        2
```

Avec liste (ou ensemble) de valeurs

```
for i in liste    # i = liste[1], liste[2]...
do
  ...
od;
```

Avec condition booléenne

```
while test   #     tant que test vaut true
do
  ...
od;
```

Pour toutes ces boucles, `RETURN` peut être utilisé pour interrompre l'exécution et quitter la procédure courante. Plus finement :

`break` sort de la boucle en court (continue l'exécution juste après le `od`).

`next` commence la prochaine itération (continue l'exécution juste avant le `od`).

8. Quelques astuces en vrac

Substitutions : `subs` (syntaxique, `x` → `y`, ne marche que sur un sous-objet qu'on peut obtenir avec une suite d'instructions `op`), `algsubs` (algébrique, `P(x)` → y, `P` polynôme). Un exemple :

```
> a := {x=1, y=1}, {x^2=2, y=3};
> subs(a[1], [x,y]);
> subs(a[2], [x,y]);
> algsubs(a[2], [x^3,y]);
```

Convertir une liste en somme / produit :

```
> convert( [x, y, z], `+` );
> convert( [x, y, z], `*` );
> convert( seq(i, i = 1..200), `*` );
```

[*voir aussi* `mul(i, i = 1..200)` *ou* `add`]

[3] en supposant $min \leqslant max$, la valeur maximale atteinte est $min + pas \left\lfloor \frac{max - min}{pas} \right\rfloor$, où $\lfloor x \rfloor$ désigne la partie entière. Il est agréable de pouvoir utiliser *max* sans se poser de question !

Quelques commandes utiles : `map` (appliquer une fonction a tous les opérandes d'un objet, i.e. les éléments d'une liste), `collect`/`expand` (pour réorganiser des polynômes), `simplify`.

Pour la partie entière et ses variantes, voir `floor`, `ceil`, `round`.

Assignations multiples : L'opérateur `:=` permet les assignations multiples pour deux séquences de noms et de valeurs de même cardinal :

```
> x,y := 1,2 ;
```

Les assignations multiples sont simultanées :

```
> x,y := y,x ;
```

échange les valeurs de `x` et `y` et a le même effet que le traditionnel

```
tmp := x ; x := y ; y := tmp ;
```

Pour corriger un programme : `printlevel := 100 ;` (par exemple), fait afficher toutes les étapes de l'exécution d'une fonction (appels de sous-fonctions, affectations de variables). Plus `printlevel` est élevé, plus il y a d'informations (5 = squelette, 1000 = tout). Plus généralement, voir `trace` et l'aide en ligne de `debugger`.