

Introduction à MAPLE⁰

1. INTRODUCTION

MAPLE¹ est un logiciel de calcul formel puissant et d'utilisation assez 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.

Nous allons survoler divers aspects de MAPLE dans les pages suivantes, afin d'être prêts sans tarder à aborder les problèmes mathématiques qui nous concerneront. Pour chaque nouvelle commande, il est conseillé de

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

Travail autonome demandé :

☞ lire ce qui suit, répondre aux questions en appliquant les conseils (1) et (2) précédents
☞ ouvrir également le fichier `introduction.mws` et exécuter la feuille de calcul MAPLE en question (Menu Edit → Execute → Worksheet) ; observer, prévoir et expliquer.

2. PRISE EN MAIN

2.1. **Prompt.** L'invite `>` (prompt) indique que MAPLE attend une instruction.

2.2. **Exécuter.** Chaque instruction se termine par `;` (ou `:` si on ne veut pas afficher le résultat). Un retour chariot (**Enter**) déclenche l'exécution [*en oubliant le point virgule ?*]. Appuyer simultanément sur **Shift** et **Enter** passe à la ligne *sans* valider la commande (pratique pour programmer lisiblement).

2.3. **Aide en ligne.** S'obtient par `?commande`

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

2.5. **Commentaire.** Taper `#blabla` (le reste de la ligne est ignoré)

2.6. **Dernière expression évaluée.** Taper `%` (l'avant dernière est `%%`, etc.).

⁰D'après B. Perrin-Riou et K. Bellabas, module MAO, Orsay

¹pour «érable» ou M A thematical PLEasure.

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 [`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 ou moins complètement une expression. Essayer

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

Modifier `Digits` au préalable [*par exemple, Digits :=50*]. A quoi correspond `pi`? Comment obtenir une valeur approchée de $\sin(\pi/13)$? Que se passe-t'il pour $\sin(\pi/3)$? [*pour certaines fonctions, MAPLE maintient une table de valeurs spéciales, op(4, eval(f)) affiche celle de la fonction f*]

4. VARIABLES

4.1. Affectation. `x := 1` pour '`x` reçoit 1'. Le nom (identificateur) de la partie gauche est associé à la valeur de la partie droite (après avoir évalué cette dernière).

4.2. Quelques pièges. Entre autres :

- 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;
```

Un résultat est complètement évalué² (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).

4.3. Libération-réaffectation. Une expression entre *côtes* libère la variable ou 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+1;
> restart; x := y; y := 'x'; x+1;
```

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)
```

²s'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;`

```
> printf("la valeur de x est %a", x)
```

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

5. QUELQUES OBJETS UTILES

5.1. **Intervalles.** Par exemple `1..100`

5.2. **Séquences (*sequence*).** Il s'agit d'une 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 à initialiser une liste d'objets identiques `1 $ 100` ou un intervalle d'entiers `$ 1..100`. La commande `seq` protège ses arguments contre une évaluation prématurée, `$` non ; en conséquence son emploi est souvent plus délicat : comparer

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

5.3. **Listes (*list*).** Une liste est de la forme `[sequence]`. C'est une séquence vue comme un seul objet, et non comme un agrégat. Définir une liste `L` contenant les entiers de 1 à 9 ordonnés par ordre croissant.

Pour extraire le i -ème élément d'un tel objet : `L[i]`. On peut affecter des valeurs `L[i] := 1` (si la liste a plus de 100 éléments MAPLE refusera les affectations [*essayer*] : utiliser un `array` au lieu d'une liste).

Le nombre d'éléments est donné par `nops(L)`, et les éléments eux-mêmes, c'est-à-dire la séquence sous-jacente, par `op(L)`. Ainsi pour ajouter à la liste précédente le nombre 10 suffit-il d'écrire `L := [op(L), 10]`. On peut utiliser `op(i, L)` au lieu de `L[i]`. On peut extraire des intervalles : `op(3..10, L)`.

Réfléchir enfin à la différence entre `f(liste)` et `f(séquence)`.

5.4. **Ensembles (*set*).** Un ensemble est de la forme `{ sequence }`. C'est une liste non ordonnée (en particulier les éléments en double sont éliminés). Définir un ensemble `E` contenant les entiers de 1 à 9. Rajouter 10 en s'inspirant de ce qui a été fait pour les listes, puis à l'aide de la commande `union`. Répéter l'ajout du nombre 10, puis le soustraire à l'aide de la commande `minus`.

Pour extraire le i -ème élément d'un tel objet : `L[i]` toujours. Attention : l'ordre des éléments d'un ensemble est arbitraire et peut changer d'une session sur l'autre. Les commandes `nops`, `op` produisent le même effet que précédemment.

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))`. Note : 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; (!!!)*].

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 `->` : par exemple `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 ?*] Alors `F(x, y)` est une expression formelle égale à `f`. Un raccourci si l'expression existe déjà : `F := unapply(f,x,y)`;

6.1. **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. MAPLE refusera qu'un argument de `f` soit modifié dans le corps de celle-ci. Ainsi

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

produit l'erreur `Illegal use of a formal parameter`.

6.2. **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 ?*]. Vérifier avec `op(4, eval(f))`;

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).

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);
```

```
> plot(pi,1..100);
> pi1 := x -> nops(select(isprime, [seq(i,i=1..x)]));
> plot(pi1(x),x=1..100);
```

7. PROGRAMMATION

```
> if test1 then      # si ... alors
>   ...
> elif test2 then   # sinon si ... alors
>   ...
> elif testn then
>   ...
> else                # sinon
>   ...
> fi;
```

7.1. Branchement conditionnel. 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 :

- $x \leq 0$ (pour \leq)
- $x \neq y$ ou bien $\text{not}(x = y)$ (pour $x \neq y$)
- $(\text{type}(p, \text{integer}) \text{ and } \text{isprime}(p))$.

7.2. 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’indice³. Si $\text{min} > \text{max}$, la boucle n’est pas exécutée.

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,

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**).

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

8. QUELQUES ASTUCES EN VRAC

8.1. **Substitutions.** Commandes `subs` (syntaxique, $x \rightarrow y$, ne s'applique que sur un sous-objet qu'on peut obtenir avec une suite d'instructions `op`), `algsubs` (algébrique, $P(x) \rightarrow y$, P polynôme). Un exemple :

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

8.2. **Convertir une liste en somme / produit.** Par exemple :

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

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

8.3. **Quelques commandes utiles.** Commandes `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`, `normal`.

8.4. **Pour corriger un programme.** La commande `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`.

RÉFÉRENCES

- [C-T] J.-M. CORNIL, P. TESTUD, *Maple, introduction raisonnée à l'usage de l'étudiant, de l'ingénieur et du chercheur*, Springer, Berlin, 1995.
- [D-G] PH. DUMAS, X. GOURDON, *Maple, son bon usage en mathématique*, Springer, Berlin, 1997.
- [P-R] B. PERRIN-RIOU, *Algèbre, arithmétique et maple*, Cassini, Paris, 2000.