Introduction à Maple

1re partie : Utilisation de Maple

Pierre BÉJIAN
bejian@free.fr
http://bejian.free.fr

16 septembre 2002

Table des matières

Introduction

Maple est avant tout un logiciel qui permet de faire des calculs « formels », c'est à dire des calculs avec des expressions symboliques, des variables abstraites. Mais il est aussi possible de faire des calculs numériques et des graphiques en 2 ou 3 dimensions.

Maple est un logiciel payant dont une version de démonstration (limitée en mémoire) est disponible à l'adresse ftp://ftp.maplesoft.com/pub/maple/demo/windows/mvr4demo.exe.

Ce document a pour but de permettre au lecteur d'acquérir les notions de bases du logiciel afin d'aborder des séances de Travaux Pratiques dans de bonnes conditions. Il s'agit en fait de la première partie d'un cours qui en contient deux. Les fonctionnalités mathématiques seront abordées dans le second document. Ce cours est aussi disponible sous forme électronique à l'adresse http://bejian.free.fr (section Maple).

Cette présentation n'est en aucun cas exhaustive, pour plus de détails vous pouvez consulter l'aide en ligne ou bien un des nombreux livres existants sur le sujet.

1 Utilisation de Maple

1.1 Interface et prise en main

Au lancement du logiciel une *feuille de calcul* (worksheet) est ouverte par défaut. En haut de celle ci figure le *prompt* (>) symbolisant le fait que Maple attend une *ligne de commande*. Par exemple :

```
> 1+2;
```

$$3$$

En validant par la touche *Entrée* on obtient le résultat centré sur la ligne suivante. Notez bien le point-virgule « ; » en fin de ligne. Si on met un deux-points « : » à la place, Maple fait le calcul mais n'affiche pas le résultat. Cela est souvent utilisé lors de calculs intermédiaires. On peut aussi effectuer plusieurs calculs sur la même ligne :

```
> 25+2; 3+4:  4-1;
```

$$27$$
$$3$$

Si on veut passer à la ligne sans effectuer le calcul on utilise la combinaison de touches *Shift* + *Entrée*. Cela peut servir lorsqu'une commande tient sur plusieurs lignes.

On peut placer des commentaires en utilisant le caractère #. Tout ce qui suivra, jusqu'à la fin de la ligne, ne sera pas évalué par Maple.

```
> 7*(2+1); # A-t-on vraiment besoin de Maple pour faire un tel calcul ?
```

$$21$$

Lorsqu'un calcul semble ne pas se terminer on peut l'interrompre à l'aide du bouton STOP de la barre d'outils.

On peut à tout moment utiliser l'aide en ligne du logiciel (menu *help*). Si on cherche des précisions sur l'utilisation d'une commande particulière, par exemple la fonction `sqrt` (racine carrée), on tape :

```
> ?sqrt
```

Pour utiliser le dernier calcul effectué (mais pas nécessairement affiché) on utilise le symbole « % » [1] (`%%` pour l'avant dernier et `%%%` pour l'antépénultième).

Il faut faire très attention à la chronologie des calculs. Une *session* Maple est l'ensemble de tout ce qui s'est passé depuis le lancement du logiciel, que ce soit les *entrées* (commandes tapées par l'utilisateur), les *sorties* (résultats affichés à l'écran) ou les opérations non affichées (par exemples des calculs intermédiaires se terminant par un « : »). L'utilisateur peut à tout moment revenir au début de sa feuille de calcul et revalider une des lignes de commandes. A cet instant il n'y a plus concordance entre la chronologie réelle et l'ordre apparent des calculs à l'écran. Par exemple la commande `%` ne donnera plus le même résultat ! Ce point est très important car il est une source d'erreurs assez fréquentes.

La feuille de calcul ne contient que les *entrées* c'est à dire les commandes tapées par l'utilisateurs. Celle ci peut être sauvegardée sous forme de fichier .MWS (pour Maple WorkSheet) ou bien exportée sous forme de simple fichier texte. Lors de l'ouverture d'une feuille de calcul il est nécessaire de revalider l'ensemble des lignes de commandes pour faire coïncider l'état réel du logiciel avec l'affichage de la feuille de calcul.

[1] Attention, jusqu'à la version VR4 c'est le symbole « " » qui était utilisé pour rappeler le dernier calcul. Ce dernier sert maintenant à délimiter les chaînes de caractères.

1.2 Expressions et variables

Affectation

L'opération consistant à stocker une expression dans une variable est l'*affectation* et utilise le symbole « := ». On écrira par exemple

```
> x:=2;
```

pour donner la valeur 2 à la variable *x*.

Réinitialisation

On peut ensuite « vider » la variable de sa valeur en la réinitialisant par l'une des deux commandes :

```
> unassign('x');
```

ou bien

```
> x:='x';
```

Pourquoi les apostrophes ? La raison est importante à comprendre : il faut bien distinguer une variable (que l'on peut considérer comme une case mémoire) de sa valeur. Tant qu'une variable n'est pas affectée, elle représente un symbole, une variable abstraite. Sa valeur est alors simplement son nom. Mais dès qu'une variable contient une valeur (par exemple un nombre entier), Maple remplace toute occurrence de celle-ci par son contenu. Ainsi si on désire parler de *la variable x* et non pas de *l'expression stockée dans la variable x*, on « protège » celle-ci en entourant son nom par des apostrophes.

Pour réinitialiser toutes les variables on utilise la commande `restart`. Il est préférable de l'utiliser quand on commence un nouveau travail, indépendant de ce qui précède (par exemple une nouvelle feuille de calcul).

Nom d'une variable

Notons pour commencer que Maple fait la différence entre majuscules et minuscules. Un nom de variable commence en général par une lettre et peut contenir des chiffres et des lettres (non accentuées). Il peut aussi contenir le caractère de soulignement « _ ». Par exemple :

```
> Nombre_de_solutions:=2;
```

En fait un nom de variable peut contenir des caractères plus exotiques (et même des espaces) à condition de « protéger » celui-ci par des accents graves (AltGr+7). Par exemple :

```
> `Nombre de solutions de l'équation (*)`:=2;
```

Mais cet usage n'est pas recommandé pour des raisons de lisibilité.

Pour terminer précisons qu'un nom ne peut pas dépasser 499 caractères et que l'on ne peut pas utiliser comme nom de variable un mot déjà utilisé en interne par Maple (par exemple Pi, infinity, sqrt, ...).

1.3 Types de données numériques et symboliques

La plupart des expression Maple ont un *type* qui caractérise leur nature. Il y a, par exemple, plusieurs types numériques (entier, rationnels, flottant, ...) ainsi que le type *symbolique*.

On peut demander à Maple le type d'une expression par la commande :

```
> whattype(3); whattype(3.0);
```

integer

float

Pour chaque type de données présenté ici nous donnons en plus de sa description, quelques fonctions qui lui sont spécifiques. Pour tous les types numériques et symboliques, les opérations arithmétiques élémentaires sont notées +, −, *, / et ^ (ou **) pour la fonction puissance.

Nombres entiers

Le type `integer` est celui des entiers relatifs. Maple peut manipuler de très grands entiers et fait avec ceux-ci des calculs exacts.

```
> 50!-2^27;
```

$$30414093201713378043612608166064768844377641568960511999865782272$$

Nombres rationnels

Le type `fraction` désigne les nombres rationnels non-entiers. Le type `rational` regroupe les types `integer` et `fraction`. Notons que Maple réduit automatiquement les fractions.

Nombres réels

En ce qui concerne les nombres réels il faut faire très attention. Le point essentiel est qu'un ordinateur ne peut pas stocker une quantité infinie d'informations. Le nombres réels sont en fait de deux catégories. Il y a d'abord le type `float` pour les approximations en virgule flottante. Ces nombres sont codés par mantisse m (généralement entre 0 et 1) et exposant e (généralement 10) sous la forme mb^e. Ces nombres réels sont donc tronqués avec une précision que l'on peut modifier en changeant la valeur de la variable `Digits`. Il faut donc absolument avoir à l'esprit que les calculs et résultats avec des nombres de type `float` sont approchés.

Mais il existe une autre catégorie de nombres réels : ceux obtenus à partir de fonctions classiques (par exemple racine carrée, fonctions trigonométriques et hyperboliques, fonctions logarithme et exponentielle, ...). Ces nombres ne sont pas évalués sauf si leur argument est déjà de type `float`. Leur résultat reste donc sous forme symbolique (comme $\sin(1)$ ou $\sqrt{2}$), et les calculs sont dans ce cas exacts.

Mais on peut « forcer » l'évaluation sous forme de `float` à l'aide de la commande `evalf` (*eval to float*). Pour obtenir une approximation de $\sin(3)$ avec 20 chiffres significatifs on tape par exemple :

```
> evalf(sin(3),20);
```

$$0.14112000805986722210$$

Il est bien sur possible lors d'un calcul de mélanger les types `integer`, `fraction` et `float`. Dans ce cas le résultat est de type `float` (donc approché).

Nombres complexes

Le nombre complexe de module 1 et d'argument $\frac{\pi}{2}$ est noté `I` en Maple (attention à la majuscule). Un nombre complexe est une expression `a+b*I`, où `a` et `b` sont rationnels ou réels (dans ce dernier cas les calculs seront donc approchés). Notons que Maple remplace directement `I`^{2} par -1.

Pour obtenir l'écriture cartésienne (`a+b*I`) d'une expression on utilise `evalc`. Les fonctions `Re`, `Im`, `abs` donnent respectivement la partie réelle, la partie imaginaire et le module d'un nombre complexe. La fonction `argument(z)` permet d'obtenir l'argument de `z` situé dans $]-\pi,\pi]$.

On peut appliquer la fonction `evalf` a un nombre complexe afin d'obtenir une valeur approchée de sa partie réelle et de sa partie imaginaire.

Constantes

Maple utilise un certain nombre de constantes symboliques. Par exemple `Pi` désigne le nombre π, `gamma` désigne la constante γ d'Euler (limite de $\sum_{k=1}^{n} \frac{1}{k} - \ln(n)$, quand $n \to \infty$) et `infinity` désigne la notion abstraite d'infinie (utile pour l'écriture des limites).

On peut par exemple obtenir la limite précédente par :

```
> limit(sum(1/k,k=1..n)-ln(n),n=infinity);
```

$$\gamma$$

Notons que l'utilisateur peut définir de nouvelles constantes symboliques en modifiant la variable `constants` (voir l'aide en ligne).

Booléens

Le type `boolean` est celui des expressions logiques. Les valeurs possibles sont `true` (vrai), `false` (faux) et `FAIL` (pour représenter une expression dont on ne connait pas la véracité).

On peut demander l'évaluation booléenne d'une expression contenant des opérateurs de relations ($<$ strictement inférieur, $>$ strictement supérieur, $<=$ inférieur ou égal, $>=$ supérieur ou égal, $=$ égal, $<>$ différent) en utilisant la fonction `evalb` :

```
> evalb(I^2=-1);
```

$$true$$

On dispose bien sûr des opérateurs logiques classiques : `not` (négation), `and` (et) et `or` (ou non exclusif)

Variables symboliques

La force de Maple est la possibilité de faire du calcul symbolique, c'est à dire avec des variables non-affectées (ne contenant aucune valeur). Une telle variable est du type `symbol`.

```
> X:=(bon + jour)/cou + (sa - 3*lut)/(1+cou); simplify(X);
```

$$X := \frac{bon + jour}{cou} + \frac{sa - 3\,lut}{1 + cou}$$
$$\frac{bon + bon\,cou + jour + jour\,cou + cou\,sa - 3\,cou\,lut}{cou(1 + cou)}$$

Rappelons qu'il existe des règles concernant les noms de variables, mais que l'utilisation des accents graves « ‘ » permet d'utiliser presque n'importe quel mot. Il est même possible d'utiliser des lettres grecques, il suffit pour cela de les taper en toutes lettres (par exemple `alpha` pour α).

Une opération intéressante est la possibilité de former des mots par *concaténation*, c'est à dire par recollement. On utilise pour cela l'opérateur « . » [2].

```
> `Bon`.`jour`;
```

$$Bonjour$$

Cette possibilité devient particulièrement intéressante quand on utilise des variables affectées (par des valeurs numériques). Les variables sont en effet évaluées avant la concaténation (sauf celle le plus à gauche).

```
> i:=4:  j:=7:  print(`A`.i.j);
```

$$A47$$

Remarque. Si `Bon` et `jour` sont non affectés mais que `Bonjour` l'est alors le résultat est le contenu de `Bonjour`.

Combiné avec la fonction `print` (affichage à l'écran) on dispose ainsi d'une manière élégante pour présenter des résultats.

```
> x:=3:  resultat:=x^2:  print(`Le carré de `.x.` est `.resultat);
```

$$Le\ carré\ de\ 3\ est\ 9$$

C'est d'ailleurs sous cette forme qu'il est préférable d'afficher les résultats obtenus.

[2] Attention, depuis la version 6 de Maple l'opérateur de concaténation est le symbole « || ».

Chaînes de caractères

Depuis la version VR5 Maple fait la différence entre les variables symboliques (type `symbol`) et les chaînes de caractères (type `string`). Il s'agit tout simplement d'une suite de caractères quelconques encadrée par le symbole « " ». Il ne s'agit plus d'un nom destiné à représenter une variable mais d'un mot au sens usuel. Les principales opérations sont les suivantes :

- la concaténation à l'aide de l'opérateur « . » ;
- la conversion d'une chaîne `s` en symbole par `convert(s,symbol)` ;
- la conversion d'un symbole `s` en chaîne par `convert(s,string)` ;
- l'extraction d'une sous-chaîne à l'aide de `substring` ;

```
> s:="Puissante est la force dans sa famille":  substring(s,3..12);
```

"issante es"

- la recherche d'un motif par la fonction `Searchtext` ;

```
> SearchText("force",s);
```

18

1.4 Types de données structurées

Après avoir fait le tour des types de base, nous allons maintenant aborder des types qui permettent de regrouper en une seule entité un grand nombre de données (typiquement un tableau).

Séquences

Une *séquence* (type `exprseq`) est une suite d'objets séparés par des virgules. Notons que ces objets peuvent être de types différents. Le nom `NULL` désigne la séquence vide. Il existe plusieurs façons de créer une séquence :

- la construction directe qui consiste à écrire tous les éléments de la séquence ;

```
> S:=1,4,9,16,25,36;
```

- en *concaténant* des séquences déjà existantes

```
> S1:=NULL: S2:=1,4,9:  S3:=16,25,36:  S:=S1,S2,S3;
```

S:=1,4,9,16,25,36

- l'utilisation de la commande `seq` ;

```
> S:=seq(k^2,k=1..6);
```

Remarque. Notons que la variable `k` peut tout a fait être déja utilisée. Dans ce cas `k` retrouve sa valeur d'origine après l'exécution de `seq`.

On peut ensuite accéder au i^e élément de la séquence `S` par `S[i]`.

Les séquences sont utilisées par exemple pour donner les solutions d'une équation. Néanmoins, pour des raisons dues au fonctionnement interne du logiciel on préfère la notion de *liste* à celle de *séquence*.

Listes

Une *liste* (type `list`) est une séquence délimitée par les caractères `[` et `]`. Ainsi si `S` est une séquence, `[S]` est la liste correspondante. Dans l'autre sens, si `L` est une liste alors `op(L)` est la séquence sous-jacente. La liste vide est désignée par `[]`. Voici quelques opérations importantes concernant les listes :

- `nops(L)` donne le nombre d'élément de la liste `L` ;
- `L[i]` (ou `op(i,L)`) donne le i^e élément de le liste `L` (pour i entre 1 et `nops(L)`) ;
- `op(i..j,L)` donne la séquence des éléments de `L` du i^e au j^e (pour $i \leq j$ entre 1 et `nops(L)`) ;
- `L[i]:=x` remplace le i^e élément de la liste `L` par `x` ;
- `member(x,L)` vaut `true` si `x` appartient à la liste `L` (`false` sinon) ;
- `convert(L,multiset)` permet de connaitre le nombre d'occurrence de chaque élément en donnant la liste des [*elem*,*nom*] où *elem* est un élément de la liste `L` et *nomb* le nombre de fois qu'il y figure ;
- `[op(L1),op(L2)]` est la concaténation des listes `L1` et `L2` ;
- `sort` permet de trier une liste (voir l'aide en ligne) ;
- `map(f,L)` permet d'appliquer la fonction `f` (voir 1.6) à tous les éléments de la liste `L` ;

```
> L:=[1,2,3]:  K:=map(exp,L);
```

$$K := [e^1, e^2, e^3]$$

- utilisation de la fonction `convert` pour faire la somme (paramètre `'+'`) ou le produit (paramètre `'*'`) de tous les éléments d'une liste ;

```
> convert(K,'+');
```

$$e^1 + e^2 + e^3$$

Remarque. Une alternative à l'utilisation de `convert('+')` est la fonction `sum` :

```
> sum(k^2,k=1..3);
```

$$14$$

Mais **attention**, contrairement au cas de la fonction `seq`, la variable `k` ne doit pas être déjà affectée. Pour plus de précautions, on prendra soin de réinitialiser l'indice de sommation (si possible) ou alors de protéger l'indice **et** l'expression à sommer par des « ' ».

```
> k:=456:  sum('k^2','k'=1..n);
```

$$\frac{1}{3}(n+1)^3 - \frac{1}{2}(n+1)^2 + \frac{1}{6}n + \frac{1}{6}$$

Il existe aussi une fonction `product`, qui s'utilise de la même façon.

Ensembles

Un *ensemble* (type `set`) est une séquence délimitée par les caractères { et }. Cela correspond bien sûr à la notion mathématique d'ensemble. Il n'y a par conséquent, dans une telle structure, ni ordre, ni répétition (contrairement au cas des listes).

```
> evalb({1,3,6,1,1}={6,1,3});
```

$$true$$

On dispose des opérations ensemblistes standards par les opérateurs `union`, `intersect` et `minus`.

```
> A:={0,1,2,3}:  B:={-3,-2,-1,0}:  A union B; A intersect B; A minus B;
```

$$\{-3, -2, -1, 0, 1, 2, 3\}$$
$$\{0\}$$
$$\{1, 2, 3\}$$

Si on utilise plusieurs de ces opérateurs dans une même opération, on fera attention aux parenthèses. Pour savoir si un élément `x` est dans l'ensemble `E`, on utilise `member(x,E)`. Enfin, comme pour les listes, on peut appliquer une fonction `f` à tous les éléments de l'ensemble `E` à l'aide de `map(f,E)`.

Tables et noms indexés

Une table (type `table`) est la mise en correspondance d'un ensemble d'*indices* (qui peuvent être des expressions, ou des séquences d'expressions) et des *valeurs*. Ces valeurs sont des expressions Maple qui peuvent être de nature complexe (séquence, liste, table, procédure, ...). On dispose de plusieurs méthodes pour construire une table :

- de manière explicite, en utilisant la commande `table` ;

```
> T:=table([2=45,5=678,(4,3)=61,Aragorn=1000]);
```

$$T := \text{table}([2 = 45, 5 = 678, \mathit{Aragorn} = 1000, (4, 3) = 61])$$

- la seconde méthode consiste a utiliser directement un *nom indexé* c'est à dire un objet du type *nom*[*indice*] et d'en définir les valeurs (une table nommée *nom* est alors créée implicitement) :

```
> P[1]:=ithprime(1),P[2]:=ithprime(2),P[3]:=ithprime(3);
```

$$P_1 := 3$$
$$P_2 := 5$$
$$P_3 := 7$$

Une fois qu'une table a été créée, on peut effectuer les opérations suivantes :

- utiliser des valeurs de la table dans un calcul :

```
> T[1] - T[2] + 2*T[4,3] + T[Aragorn]-sqrt(3)*T[Galadriel];
```

$$T_1 + 1077 - \sqrt{3}\, T_{\mathit{Galadriel}}$$

- rajouter des éléments (ou modifier ceux déjà existants) à l'aide de la commande *nom*[*indice*]:=*valeur* ;
- afficher la totalité d'une table par la commande `eval` ;
- afficher l'ensembles des indices (resp. des valeurs) par l'instruction `indices` (resp. `entries`) (attention, on ne connait pas à l'avance l'ordre d'affichage des éléments mais cet ordre est le même pour les deux fonctions).

```
> indices(T); entries(T);
```

$$[2], [5], [\mathit{Aragorn}], [4, 3]$$
$$[45], [678], [1000], [61]$$

Si un élément de la table n'a pas de valeur (par exemple ici T_1 et $T_{\mathit{Galadriel}}$) celui-ci est utilisé comme une variable symbolique *indexée* (type `indexed`).

Tableaux, vecteurs et matrices

Tableaux, vecteurs et matrices sont des tables particulières. Mais contrairement au cas d'une table générale, les indices ne peuvent plus être quelconques.

Un tableau (type `array`) de dimension n est une table dont les indices sont des n-uplets constitués d'entiers consécutifs. Par exemple `array`(1..3,-10..10,0..5) crée un tableau à 3 dimensions, le premier indice variant de 1 à 3, le deuxième de -10 à 10 et le dernier de 0 à 5.

Une fois qu'un tableau a été créé, on peut définir la valeur des ses éléments de la même façon que pour une table générale.

```
> tab:=array(1..2,0..1):  tab[1,0]:=1:  tab[2,0]:=2:  tab[2,1]:=3:
```

Et on peut ensuite utiliser ces éléments dans un calcul :

```
> tab[1,0]-tab[1,1]+2*tab[2,1];
```

$$7 - tab_{1,1}$$

Il faut bien comprendre qu'un tableau (contrairement à une table) est un objet de taille fixée : une fois les dimensions définies, on ne peux pas rajouter ou supprimer d'éléments. Par contre, on peut bien sûr modifier les éléments déjà existants.

Une matrice (type `matrix`) de taille $n \times p$ est un tableau de dimension 2, le premier indice (indice de ligne) variant de 1 à n, et le second (indice de colonne) variant de 1 à p. On peut créer une matrice soit comme on le ferait pour un tableau, soit en donnant la liste de ses lignes. On utilise pour cela le constructeur `matrix` (`array` marche aussi) :

```
> A:=matrix([[1,2],[2,1]]); B:=array([[1,2,x],[3],[6,y],[0,0,0]]);
```

$$A := \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix}$$

$$B := \begin{bmatrix} 1 & 2 & x \\ 3 & B_{2,2} & B_{2,3} \\ 6 & y & {}_{3,3} \\ 0 & 0 & 0 \end{bmatrix}$$

Dans ce cas le nombre de colonnes est la taille de la première liste, et les suivantes doivent être de taille inférieure ou égale (certains éléments pouvant rester indéfinis).

De même un vecteur (type `vector`) de taille n est un tableau de dimension 1, l'indice variant de 1 à n et comme précédemment il existe un constructeur spécifique nommé `vector`.

```
> V:=vector([a, b, c]);
```

$$V := [a, b, c]$$

Les opérations classiques concernant les vecteurs et les matrices seront abordées dans la deuxième partie de ce cours.

1.5 Structures de contrôle

On dispose en Maple des structures de contrôle classiques : tests et boucles.

Structure conditionnelle

La syntaxe est la suivante :

if *condition* **then** *instructions* **else** *instructions*

Notons que le **else** est optionnel et que l'on peut utiliser **elif** (contraction de *else if*) dans le cas de conditions multiples.

```
> if x>0 then print('Le nombre '.x.' est positif')
  elif x<0 then print('Le nombre '.x.' est négatif')
  else print('Le nombre '.x.' est nul')
  fi;
```

Quand il y a plusieurs instructions il faut les séparer par des « ; » ou des « : » (pour être tout à fait précis ces symboles sont là pour terminer les instructions et non pour les séparer).

Itération

Pour répéter une opération on peut utiliser une boucle avec indice numérique. La syntaxe d'une telle boucle est la suivante :

for *indice* **from** *début* **to** *fin* **by** *pas* **do** *instructions* **od**

L'indication du *pas* est optionnelle : par défaut celui-ci vaut 1.

```
> for k from 0 to 5 by 2 do print(k) od;

                                    0
                                    2
                                    4
```

La variable *indice* peut être déjà affectée mais elle ne retrouve pas sa valeur après l'itération. Après l'exécution de la boucle la valeur de cette variable est $fin+1$.

Boucle `while`

On utilise une boucle `while` (*tant que*) quand on veut répéter une opération mais que l'on ne connait pas à l'avance le nombre d'itérations. Sa syntaxe est :

while *conditions* **do** *instructions* **od**

Remarque. En ce qui concerne l'affichage des calculs à l'intérieur d'une structure de contrôle, seul compte le « : » ou « ; » final. Les instructions peuvent donc être indifféremment terminées par « : » ou « ; ». Si l'on désire « forcer » un affichage, malgré un « : » final, on peut utiliser la fonction `print`.

```
> for k from 1 to 10
  do
   if isprime(k) then print(k) fi
  od:

                                    2
                                    3
                                    5
                                    7
```

1.6 Fonctions et procédures

Fonctions

Pour définir une fonction (au sens mathématique) on utilise la syntaxe suivante :

nom := *variable* -> *valeur* ;

On écrira par exemple

```
> f:=x->x^2;
```

ou bien

```
> g:=(x,y)->x^2+y^2;
```

dans le cas de plusieurs variables. Une fois définie, une fonction peut être utilisée dans un calcul :

```
> f(4); g(2,3);
```

$$16$$
$$13$$

On peut aussi définir une fonction par morceaux à l'aide de `piecewise` :

```
> f:=x->piecewise(x<0,0,x>1,0,1):  f(-2), f(1/2), f(10);
```

$$0,1,0$$

Remarque. Attention, il ne faut pas confondre *fonctions* et *expressions dépendant de variables*. Par exemple $P = x^2+7x+1$ est une expression dépendant de x (d'un point de vue mathématique c'est même un polynôme) qu'il ne faut pas confondre avec la fonction $x \mapsto x^2+7x+1$.

```
> P:=x^2+7*x+1;
```

$$P := x^2+7x+1$$

Mais il est possible de définir cette fonction à l'aide de l'expression P en utilisant `unapply` :

```
> f:=unapply(P,x);
```

$$f := x \to x^2+7x+1$$

Procédures

En réalité il n'y a pas lieu de distinguer fonctions et procédures. Plus précisemment une fonction telle que définie précédemment est un cas particulier de procédure.

Il s'agit dans tous les cas d'un petit programme prenant des *paramètres* en entrée et qui effectue ensuite un certains nombre d'instructions. Une procédure peut contenir des *variables locales*, servant aux calculs internes à la procédure ; ces variables n'ont de sens qu'à l'intérieur de celle-ci. Une procédure renvoie toujours un résultat : celui-ci correspondant par défaut à la dernière instruction ou bien à l'argument de la commande `RETURN`.

La syntaxe pour définir une procédure est la suivante :

nom := **proc**(*paramètres*)
  local *variables locales* ;
  instructions ;
end ;

Pour calculer par exemple la somme des carrés d'entiers on peut définir la procédure suivante :

```
> somme:=proc(n)
    local k,s;
    s:=0;
    for k from 1 to n do s:=s+k^2 od;
    s;
  end;
```

Remarquons que la dernière instruction ne fait aucun calcul : c'est elle qui détermine le résultat de la procédure `somme`.

```
> somme(3);
```

$$14$$

1.7 Programmation

Styles fonctionnel et impératif

Le langage de Maple mélange deux types de programmation. Il y a tout d'abord la programmation *impérative*, que l'on peut résumer comme une suite d'interactions avec la mémoire. L'opération de base d'un tel langage est l'affectation (consistant à mettre une *valeur* dans une *variable*). On retrouve ce style de programmation dans des langages comme Fortran, Pascal ou C.

Le style fonctionnel consiste en une succession de définitions et d'appels de fonctions. Celles-ci ont des arguments et retournent un résultat, le résultat d'une étape étant l'argument de la suivante. Ce type de programmation s'appuie sur un modèle théorique appelé le λ-calcul. Les langages fonctionnels les plus connus sont sans doute Lisp, ML et son dérivé Caml.

Ces deux types de programmation ont chacuns leurs avantages. Un programme impératif sera sans doute plus efficace car plus proche du langage machine. Mais le langage fonctionnel offre un haut niveau d'abstraction et permet souvent d'écrire des programmes plus courts et plus élégants.

Ces deux paradigmes de programmation s'opposent également sur la gestion de la mémoire. Elle est automatique dans le cas fonctionnel alors qu'il est souvent nécessaire d'allouer et libérer la mémoire dans le cas impératif (par exemple en C à l'aide des fonctions `malloc` et `free`).

Récursivité

Sans rentrer dans des détails théoriques, nous dirons qu'une fonction f est définie *récursivement* si sa définition a la forme suivante :

$$f(x) = \begin{cases} \textit{connu} & \text{si cas trivial} \\ g\,(f(\textit{une expression plus simple que } x)) & \text{sinon} \end{cases}$$

où g est une fonction explicite (le terme *calculable* serait plus aproprié). L'exemple le plus parlant est celui de la fonction factorielle définie par

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \times (n-1)! & \text{sinon} \end{cases}$$

et que l'on peut coder en Maple de la façon suivante :

```
> fact:=proc(n)
  if n=0 then 1
  else n*fact(n-1)
  fi
  end;
```

Pour qu'une telle définition ait un sens, il faut être sûr que les simplifications successives conduiront nécessairement au cas « trivial ». Dans le cas contraire on aurait un programme infini !

Même si un programme impératif peut être récursif, notons pour terminer que c'est le plus souvent en programmation fonctionnelle que l'on rencontre ce type de définition.

1.8 Traitement de texte intégré

Quand une feuille de calcul commence à devenir longue et complexe, on ressent le besoin de structurer celle-ci. Heureusement Maple permet une mise en page agréable avec les possibilités suivantes :

- écriture de texte simple ou de formules mathématiques (non évalué par le moteur de calcul) ;
- changement de police et de couleur ;
- introduction de titres.

Pour réaliser une mise en page on utilisera soit la barre d'outils soit les menus *Insert* et *Format*. En particulier, pour passer dans le mode formule, texte ou calcul on cliquera (respectivement) sur l'un des boutons suivants :

1.9 Pour aller plus loin : évaluation des expressions

Il faut bien comprendre le mécanisme d'évaluation des expressions Maple. Lorsque l'on demande l'évaluation d'une expression contenant des variables, Maple commence en principe par remplacer celles-ci par leurs valeurs. Si ces valeurs contiennent à leur tour des variables, le logiciel continue ainsi de manière récursive. On peut aussi demander une évaluation « par étapes », en donnant un deuxième paramètre à la fonction `eval` :

```
> x:=3*y:  y:=7:  eval(x+5,1); eval(x+5,2)
```

$$3y+5$$
$$26$$

Dans le cas où une variable n'est pas affectée, rappelons que sa valeur est simplement son nom et qu'elle est de type `symbol`. Si on protège une expression par des apostrophes, cela empêche l'évaluation des variables ; par contre les simplifications élémentaires sont tout de même effectuées.

```
> x:=2:  'x + x + 2*x';
```

$$4x$$

Mais il existe une exception à la règle précédente : l'évaluation d'une table ou d'une procédure, (plus généralement d'un objet complexe) renvoie le nom de celle-ci et non sa valeur. C'est ce que l'on appelle l'évaluation *au dernier nom*. Si l'on désire afficher la valeur d'un tel objet, il faut donc « forcer » l'évaluation à l'aide de la fonction `eval` :

```
> t:=array(1..3,[2,3,5]):  x:=t:  x; eval(x);
```

$$t$$
$$[2,3,5]$$

Dans l'exemple précédent, la commande `x:=t` ne copie pas la valeur de `t` dans la variable `x`. C'est le nom « `t` » qui est copié (et *x* devient en quelque sorte un « pointeur »). Si on modifie ensuite la valeur de `t`, `x` est aussi modifié.

```
> t:=array(1..3,[2,3,5]):  x:=t:  t[1]:=10:  eval(x);
```

$$[10,3,5]$$

Ce comportement pose évidemment un problème quand on désire faire la copie d'une variable de type table. On utilise pour cela la fonction `copy` :

```
> t:=array(1..3,[2,3,5]):  x:=copy(t):  t[1]:=10:  eval(x);
```

$$[2,3,5]$$

Attention, ce n'est pas le mécanisme d'évaluation « au dernier nom » qui empêche la copie directe d'un objet complexe, c'est plutôt le fonctionnement interne de Maple en ce qui concerne le stockage de ces objets. Ainsi on pourrait penser à la commande `x:=eval(t)` pour copier la valeur de la table `t` mais celle-ci ne résout pas notre problème.

Remarque. Ce genre de comportements de type « pointeur » peuvent aussi apparaître avec des objets simples :

```
> x:=y:  y:=4:  x; y:=5:  x;
                                    4
                                    5
```

L'évaluation de la variable *x* donne un résultat différent quand on change la valeur de *y*. C'est ici l'ordre des commandes qui est important. Lors d'une affectation, si le second membre n'a pas de valeur, c'est son nom qui est affecté à la variable du premier membre.