TP Maple 4 | Procédures — Séquences, ensembles et listes

Les procédures permettent à l'utilisateur de créer ses propres fonctions. Les listes et les tableaux de chiffres sont des formats courants en science et en ingénieurie ; citons l'exemple des données statistiques : sondages, résultats d'expériences en physique ou en médecine, mesures diverses en chimie, etc. Dans ce chapitre, nous exposerons comment définir et manipuler ce type de données sous Maple.

1. Les procédures

Revenons un instant sur un exemple déjà étudié. Le programme suivant calcule le plus grand de deux nombres réels x et y.

```
> x:=1:y:=2:if (x>y) then max:=x else max:=y: end if:max;

                                    2
```

L'inconvéniant d'un tel programme est qu'il faut modifier l'affectation de x et y avant de lancer le corps du programme...

```
> x:=5:y:=2:if (x>y) then max:=x else max:=y: end if:max;

                                    5
```

L'idéal serait utile de disposer d'une écriture semblable à celle des fonctions : définir une fonction max renvoyant $max(x, y)$ sans avoir à écrire sytématiquement le programme ou les affectations $x := \ldots$ et $y := \ldots$ en utilisant le copié-collé. C'est précisemment l'objet des procédures sous Maple.

1.1. Définition d'une procédure

Une procédure est un programme pouvant prendre un nombre variable d'arguments (comme une fonction) et renvoyant un résultat selon la règle suivante.

Règle de dernière évaluation

La valeur renvoyée par une procédure est le résultat du dernier calcul exécuté au sein du corps de programme.

La procédure suivante, prenant x et y pour arguments, calcule max(x, y).

```
> max := proc(x,y)
                  if ( x > y ) then  x
                                     else  y
                  fi:
         end:
```

Il suffit alors d'appeler la procédure « max » en des valeurs numériques de x et y...

```
> max(12.223,12.2231);
```

$$12.2231$$

...ou encore...

```
> max(-12.34,-12.31);
```

$$-12.31$$

L'usage des procédures est incontestablement *plus souple* que l'écriture de programmes sauvages !

Une procédure peut aussi ne comporter aucun argument, dans ce cas la syntaxe de l'appel est « $f()$ » pour une procédure f.

```
> f := proc()
              local i,s;
              s:=0;
              for i from 1 to 1000 do
                                    s:=s+i^2:
                                od:
       end:

> f();
```

$$333833500$$

Dans l' exemple précédent, on a utilisé des variables locales, c'est-à-dire des variables qui ne sont accessibles qu'au sein de la procédure : il est impossible d'accéder à leurs valeurs en dehors de la procédure. Si l'utilisateur souhaite manipuler une variable en dehors de la procédure, il la déclare globalement. Le lecteur méditera l'exemple suivant [1].

1. Cette procédure calcule la somme $\sum_{k=0}^{1000} k^2$.

```
> f := proc()
             local i,s;
             s:=0:
             for i from 1 to 1000 do
                                    s:=s+i^2:
                                od:
       end:

> s:=5: f(),s;

                                 333833500,5

> f := proc()
             local i;
             global s;
             s:=0:
             for i from 1 to 1000 do
                                    s:=s+i^2:
                                od:
       end:

> s:=5: f(),s;

                              333833500,333833500
```

Le lecteur retiendra la syntaxe suivante.

Syntaxe pour la création d'une procédures

nom := proc ($arg_1, \dots, arg_n$) *Variables d'entrée (ie arguments) de la procédure*
local $\ell_1, \dots, \ell_m$ **;** *Variables locales* $\ell_1, \dots, \ell_n$ *de la procédure*
global $g_1, \dots, m$ **;** *Variables globales* $g_1, \dots, g_n$ *de la procédure*
⋮
Corps du programme
⋮
end proc :

Appel d'une procédure

La ligne de commande **nom**$(x_1, \dots, x_n)$ renvoie le dernier résultat calculé par la procédure « *nom* » .

On prendra garde à ce que **Maple** traite les arguments comme des valeurs et non comme des variables : il est donc impossible, sous peine d'un message d'erreur « *Illegal use of a formal parameter* » de modifier au sein de la procédure les arguments $\arg_k$. On utilise dans ce cas des variables locales.

1.2. Procédures récursives

Une procédure récursive est un programme qui « s'appelle lui-même » au cours de son exécution. Ce type de procédure est adapté aux objets mathématiques définis par récurrence. On utilise la commande **RETURN** (notez bien les majuscules !) pour forcer la valeur de sortie de la procédure , la règle du premier paragraphe ne s'appliquera donc plus. Voici l'exemple d'une procédure calculant $n!$.

```
> fact := proc(n)
                  if n=0 then RETURN(1)
                         else RETURN(n*fact(n-1))
                  end if
          end proc:
> fact(30);

                      265252859812191058636308480000000
```

Le fonctionnement de cette procédure est simple. L'utilisateur écrit la commande **fact(30) ;** le logiciel doit alors calculer $30 \times fact(29)$, il « lance » donc la procédure pour la valeur 29 ; il doit alors calculer $29 \times fact(28)$, il « lance » donc la procédure pour la valeur 28 et ainsi de suite ... Le programme s'arrête cependant grâce au test qui affecte 1 à **fact(0)**. Finalement, on a

$$\textbf{fact}(\textbf{30}) := 30 \times \textbf{fact}(29) := 30 \times 29 \times \textbf{fact}(\textbf{28}) := \ldots := 30 \times 29 \times \ldots \times \textbf{fact0} := 30 \times \times 29 \times \ldots \times 1 := 30!$$

La dernière valeur calculée est donc bien 30!, valeur retournée par l'appel **fact(30)**.

Il est recommandé d'utiliser l'option « *remember* » dans les procédures récursives, elle augmente la vitesse d'éxecution du programme en conservant en mémoire la trace de ses calculs. Elle est incontournable lorsque la procédure fait appel à *deux niveaux*[2] de récursivité.

2. Cf. le deuxième exemple ci-dessous.

```
> time(fact(300));

                                  0.021

> fact := proc(n)
                  option remember;
                  if n=0 then RETURN(1)
                         else RETURN(n*fact(n-1))
                  end if:
          end proc:

> time(fact(300));

                                  0.020
```

Le temps d'exécution d'une procédure dépend clairement de la version choisie ainsi que du matériel utilisé.

1.3. Exercices

Exercice 1.

Ecrire deux procédures **Suite(n)** et **Suiter(n)**, respectivement non récursive et récursive, prenant en argument un entier n et renvoyant le terme u_n le la suite définie par

$$u_0 = 1,\ \ u_1 = 1,\ \ u_2 = 3,\ \ \forall n \in \mathbb{N},\ \ u_{n+3} = 2u_{n+2} + u_{n+1} - u_n$$

2. Les ensembles

Les ensembles sont les structures fondamentales des Mathématiques. Maple permet de définir un type **set** (ensemble) et d'effectuer des opérations élémentaires portant sur des objets de ce type.

2.1. Définition et commande member

Un ensemble est *une énumération entre accolades d'expressions séparées par des virgules.*

```
> ens1:={1,5,4,78,2154,646};
```

$$ens1 := 1,4,5,78,2154,646$$

```
> ens2:={45,865,5,78,2154};
```

$$ens2 := 5,45,78,2154,865$$

Conformément à la définition mathématique, les éléments ne sont pas ordonnés et chaque élément ne figure qu'une seule fois.

```
> e:={x,y,x,x,y,z};
```

$$\{x,z,y\}$$

On peut tester l'appartenance d'un élément à un ensemble au moyen de la commande **member** ; **member(a,E)** renvoie *true* (vrai) si $a \in E$ et *false* sinon.

```
> member(4,ens1), member(4,ens2);
```

true, false

Remarquons qu'un ensemble est une expression dont les opérandes sont les éléments [3].

```
> op(ens1), nops(ens2);
```

$$1,4,5,78,2154,646,5$$

2.2. Opérations ensemblistes

Maple permet la programmation des opérations élémentaires sur les ensembles :

3. Attention, l'ordre des éléments d'un ensemble est imprévisible !

```
> ens1 union ens2;
```

$$1,4,5,45,78,2154,646,865$$

```
> ens1 intersect ens2;
```

$$5,78,2154$$

```
> %% minus %;
```

$$1,4,45,646,865$$

Manipulation des ensembles

- **member(x,E)** : renvoie *vrai* si x appartient à E.
- **A union B** : renvoie $A \cup B$.
- **A intersect B** : renvoie $A \cap B$.
- **A minus B** : renvoie $A \setminus B$.
- **nops(E)** : renvoie le cardinal de E.

2.3. Exercices

Exercice 2.

Soient E et F deux ensembles finis non vides et f une application de E dans F. Ecrire une procédure **ImageReciproque(B)** d'argument B (une partie de F) renvoyant l'ensemble $f^{-1}(B)$. Tester cette procédure sur un exemple de votre choix.

3. Les séquences

Nous exposerons ici la structure de *séquence* (parfois appelée *suite*). Les séquences sont moins importantes que les listes (voir le paragraphe suivant) mais interviennent souvent au titre d'intermédiaires de calcul comme nous le remarquerons plus loin dans ce cours.

3.1. Définition

Une séquence sous Maple est *une énumération d'expressions séparées par des virgules.*

```
> seq1:=1,65,87,101,12354;
```

$$seq1 := 1,65,87,101,12354$$

```
> seq2:=x,x^2,d-x;
```

$$seq2 := x, x^2, d-x$$

Contrairement aux ensembles, les redondances et l'ordre des éléments sont importantes lors de la définition d'une séquence. On comparera ce qui suit avec les exemples du paragraphe sur les ensembles.

```
> s:=1,2,2,2,3,4,3,5;
```

$$1,2,2,2,3,4,3,5$$

3.2. Opérations élémentaires

Les séquences sont concaténables, c'est-à-dire *juxtaposables*. La syntaxe correspondant à cette opération est simple et naturelle.

```
> seq3:=seq1,seq2,45,u;
```

$$seq3 := 1,65,87,101,12354,x,x^2,d-x,45,u$$

Comme les variables, on peut évaluer les séquences.

```
> x:=2:seq3;
```

$$1,65,87,101,12354,2,4,d-2,45,u$$

On peut isoler les éléments d'une séquence.

```
> seq3[7], seq3[10];
```

$$4,u$$

Signalons en conclusion que les séquences ne sont accessibles qu'en seule lecture.

```
> seq3[1]:=1;
Error, cannot assign to an expression sequence
```

3.3. La commande seq

La commande **seq** permet à l'utilisateur de réaliser des listes à partir d'une formule générale.

```
> seq4:=seq(2/k,k=1..11);
```

$$\text{seq4} := 2, 1, \frac{2}{3}, \frac{1}{2}, \frac{2}{5}, \frac{1}{3}, \frac{1}{7}, \frac{1}{4}, \frac{2}{9}, \frac{1}{5}, \frac{2}{11}$$

Attention ! La variable k doit être non-affectée et l'intervalle de variation de k doit être un intervalle fixé d'entiers.

On peut également employer le symbole « $ » pour construire des séquences. Dans ce cas, les variables non-affectées sont autorisées aux bornes de variation de k.

```
> seq5:=2/k $ k=1..n:eval(subs(n=11,seq5));
```

$$2, 1, \frac{2}{3}, \frac{1}{2}, \frac{2}{5}, \frac{1}{3}, \frac{1}{7}, \frac{1}{4}, \frac{2}{9}, \frac{1}{5}, \frac{2}{11}$$

```
> seq6:=$1..5;
```

$$\text{seq6} := 1, 2, 3, 4, 5$$

4. Les listes

Les listes sont des objets essentiels. Elles permettent notament le traitement de séries de résultats en ingénieurie.

4.1. Listes vs. séquences

Une liste est *une énumération d'expressions entre crochets séparées par des virgules.*

```
> list1:=[1,3,45,875,964];
```

$$list1 := [1,3,45,875,964]$$

```
> list2:=[seq(k,k=1..14)];
```

$$list2 := [1,2,3,4,5,6,7,8,9,10,11,12,13,14]$$

Puisqu'une liste n'est ni plus ni moins qu'une séquence entre crochet, on pourra créer de la sorte une liste.

```
> list3:=[seq4];
```

$$\left[2,1,\frac{2}{3},\frac{1}{2},\frac{2}{5},\frac{1}{3},\frac{1}{7},\frac{1}{4},\frac{2}{9},\frac{1}{5},\frac{2}{11}\right]$$

4.2. Opérations sur les listes

Contrairement aux séquences, mais à l'instar des ensembles, les listes sont des expressions.

Liste et expression

La liste

$$[a_1, a_2, \ldots, a_n]$$

est considérée par Maple comme une expression dont les opérandes sont : $a_1, a_2, \ldots, a_n$.

Ainsi, la commande **op** retourne-t-elle la séquence formée par les éléments d'une liste.

```
> op(list3);
```

$$2,1,\frac{2}{3},\frac{1}{2},\frac{2}{5},\frac{1}{3},\frac{1}{7},\frac{1}{4},\frac{2}{9},\frac{1}{5},\frac{2}{11}$$

Allons plus loin : on peut fusionner des listes en passant par la concaténation des séquences correspondantes.

```
> list4:=[op(list1),op(list3)];
```

$$liste4 := \left[1,3,45,875,964,2,1,\frac{2}{3},\frac{1}{2},\frac{2}{5},\frac{1}{3},\frac{2}{7},\frac{1}{4},\frac{2}{9},\frac{1}{5},\frac{2}{11}\right]$$

Les listes sont acessibles en lecture et en enregistrement (contrairement aux séquences qui ne sont accessibles qu'en seule lecture) : on peut modifier une liste au cours d'un programme.

```
> list4[1]:=0:list4;
```

$$\left[0,3,45,875,964,2,1,\frac{2}{3},\frac{1}{2},\frac{2}{5},\frac{1}{3},\frac{2}{7},\frac{1}{4},\frac{2}{9},\frac{1}{5},\frac{2}{11}\right]$$

Les commandes **subs** et **subsop** permettent respectivement l'évalutation d'une liste et la modification de certains termes.

```
> L:=[x,x+y,2*x]:
> subs(x=3,L);
```

$$[3,3+y,6]$$

```
> subsop(3=Z,L);
```

$$[x,x+y,Z]$$

Voici un aperçu non exhaustif de commandes pour la manipulation des listes sous Maple...

Manipulation des listes

- **nops(L)** : renvoie le nombre d'éléments de la liste L.
- **[op(p..q,L)]** : renvoie la sous-liste des éléments indexés de p à q de L.
- **[op(L1),op(L2)]** : fusionne les deux listes $L1$ et $L2$.
- **subs(x=a,L)** : évalue L en $x = a$.
- **subsop(L,i=A)** : affecte la valeur A à $L[i]$.
- **select** : permet de sectionner des termes d'une liste selon un critère.
- **sort** : permet de trier une liste par ordre croissant des termes lorsque cela a un sens.

4.3. Exercices

Exercice 3.

Ecrire une procédure **Inverse(L)** prenant en argument une liste **L** et renvoyant la liste obtenue en balayant **L** dans le sens décroissant des indices. On proposera deux solutions : avec et sans boucle (en utilisant **seq**).

Exercice 4.

Ecrire une procédure **TestCroissante(L)** prenant en argument une liste de nombres réels **L** et testant si elle est croissante. On proposera deux solutions : l'une utilisant **sort**, l'autre en utilisant une boucle.

Exercice 5.

Ecrire un procédure **Doublons(L)** prenant en argument une liste de nombres réels **L** et testant si elle comptient un terme apparaissant au moins deux fois. On proposera deux solutions : en effectuant deux boucles imbriquées *puis* en utilisant astucieusement la commande **nops**.

Exercice 6.

Ecrire une procédure **Signes(L)** d'argument une liste L renvoyant le nombre de changements de signe d'une liste de réels donnés. On adoptera la définition suivante : le nombre de changement(s) de signe dans $(u_0,\dots,u_n)$ est égal au nombre de changements de signe dans la suite obtenue à partir de $(u_1,u_2,\dots,u_n)$ en supprimant les termes nuls. Par convention $CS(\varnothing)=0$.

5. Tableaux

Les tableaux sont des objets incontournables en informatique ; pour s'en convaincre il suffit de penser à la manipulation de données statistiques (cf. traitement du signal pour le son, l'imagerie, etc.) ou encore aux problèmes de codage (cf. compression des données).

5.1. La commande Array

Maple permet de ranger des expressions dans un *tableau* au moyen de la commande **Array**[4]. Un *tableau* n'est autre qu'une collection de variables partageant le même préfixe et indexés par un nombre quelconque d'indices. Dans le cas où il n'y a qu'un seul indice, un tableau T peut être représenté par une succession de variables :

$$T[1],T[2],T[3],T[4],T[5]$$

Dans le cas où il n'y a que deux indices, on peut assimiler un tableau à un quadrillage où chacune des cases est une variable.

4. Attention à ne pas la confondre avec l'ancienne commande **array** (sans majuscule) encore valable sous Maple 11 pour des raisons de compatibilité mais pour laquelle tout ce qui suit n'est pas valable.

$$\begin{bmatrix} T[1,1] & T[1,2] & T[1,3] & T[1,4] \\ T[2,1] & T[2,2] & T[2,3] & T[2,4] \\ T[3,1] & T[3,2] & T[3,3] & T[3,4] \end{bmatrix}$$

La variable $T[i,j]$ se trouve dans la case repérée par la ligne i et la colonne j.

On pourra créer un tableau par la commande [5] **Array(*dimensions*, *contenu*)** où ***dimensions*** est de la forme $1..n, 1..p$ (n=nombres de ligne(s), p=nombre de colonne(s)) [6] et où ***contenu*** peut être vide ou contient une liste (donc entre crochets) de lignes écrites sous forme de listes (donc aussi entre crochets).

```
> Array(1..1,1..2,[1,5]), Array(1..2,1..2,[[1,0],[0,1]]);
```

$$[1\ 5],\ \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Par défaut, les éléments non spécifiés d'un tableau sont nuls.

```
> T:=Array(1..2,1..2);
```

$$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

5.2. Lecture et modification d'un tableau

Les tableaux sont accessibles en lecture et en enregistrement [7] après leur affectation.

```
> T:=Array(1..2,1..2): T[1,2]:=3: T[2,1]:=1:
```

$$\begin{bmatrix} 0 & 3 \\ 1 & 0 \end{bmatrix}$$

5. L'objet ainsi créé est considéré par le logiciel comme une expression et on peut donc utiliser la fonction **op** pour obtenir des renseignements sur un tableau. Nous ne détaillerons pas ce point beaucoup plus technique.
6. En fait, plus généralement, **Array** autorise des indices *entiers relatifs*.
7. C'est-à-dire que l'on peut modifier les cases à volonté au cours d'un programme.

Manipulation des tableaux

► **T :=Array(1..n,L)** : crée un tableau à 1 ligne et n colonnes à partir les valeurs de la liste L de longueur n.

► **T :=Array(1..n)** : crée le tableau nul à 1 ligne et n colonne(s).

► **T :=Array(1..p,1..q)** : crée le tableau nul à p ligne(s) et q colonne(s).

► **T[i] (respect. T[i,j])** : renvoie le terme d'indice i de T (respect. d'indices i, j).

► **T[i] :=a (respect. T[i,j] :=a)** : affecte la valeur a au terme d'indice i de T (respect. d'indices i, j).

6. Fonctions agissant sur une liste ou un tableau

On peut appliquer à chaque coefficient d'une liste ou d'un tableau une fonction donnée.

6.1. Commandes map et zip

La syntaxe est la suivante :

La fonction **map**

map(*nom de la fonction, nom de l'objet*)

```
> L:=[0,Pi/2,Pi]: map(sin,L);
```

[0, 1, 0]

On peut également appliquer à deux objets une fonction de deux variable au moyen de la commande **zip**.

La fonction **zip**

zip(*nom de la fonction, nom de l'objet*$_1$*, nom de l'objet*$_2$)

```
> L:=[0,-1]: K:=[1,1]: f:=(x,y)->x+y: zip(f,L,K);
```

[1,0]

Ces fonctions s'étendent aux vecteurs et aux matrices.

6.2. Exercices

Exercice 7.

Créer la liste X des fractions de la forme $\frac{k\pi}{10}$ avec $0 \leqslant k \leqslant 10$ puis la liste Y de leurs sinus. Calculer une valeur approchée de

$$S = \frac{\pi}{10} \sum_{k=0}^{10} \sin\left(\frac{k\pi}{10}\right).$$

Exercice 8.

Créer la liste A des vingts premiers entiers de deux manières différentes : au moyen de la commande **seq** puis en utilisant \$. Créer la liste B dont les éléments sont les carrés de la liste A.