

TP Maple 4 | Éléments de programmation

Les structures de branchement (tests) et de répétition (boucles) sont au fondement de la programmation informatique. Elles permettent respectivement d'effectuer certaines tâches en fonction de conditions choisies par l'utilisateur et d'effectuer une tâche à plusieurs reprises. Les procédures permettent à l'utilisateur de créer ses propres opérateurs sur divers objets.

1	Tests	1
1.1	Les expressions booléennes	1
1.2	La structure de test if ... then	2
2	Boucles	3
2.1	La boucle for	3
2.2	La boucle while	4
2.3	Exemple : calcul d'une valeur approchée	5
3	Les procédures	6
3.1	Définition d'une procédure	6
3.2	Procédures récursives	8
4	Exercices	9

1. Tests

1.1. Les expressions booléennes

Par définition, une variable booléenne ne prend que deux valeurs : vrai (true) ou faux (false). Le logiciel manipule ce type de variables. On pourra comparer des expressions de même type à l'aide des opérateurs suivants.

Opérateurs de comparaison

▶ = : signe = (égalité au sens mathématique).	▶ <> : signe ≠.	▶ >= : signe ≥.	▶ > : signe >.
	▶ <= : signe ≤.	▶ < : signe <.	

Il faut prendre garde au fait que Maple n'évaluera ce type d'expressions que si l'utilisateur lui demande par l'intermédiaire de la commande d'évaluation des booléens **evalb**, qui renvoie **false** ou **true** selon la validité de la proposition :

L'exemple précédent utilise deux environnements de test. On peut le programmer en un seul avec la commande `else` qui signifie « sinon ».

Syntaxe d'un test (II)

```
if condition then groupe 1 else groupe 2 fi;
```

où *groupe 1* est un ensemble de commandes à effectuer si la *condition* est vérifiée et *groupe 2* est un ensemble de commandes à effectuer dans le cas contraire.

```
> a:=4: if a<3 then 'a est strictement inférieur à 3' else 'a est supérieur à 3' fi;
```

```
a est supérieur à 3
```

Dans le cas où il y a plus de deux conditions, on utilisera la syntaxe indiquée ci-dessous où la commande `elif` est une contraction de « else-if ».

Syntaxe d'un test (III)

```
if condition 1 then groupe 1
elif condition 2 then groupe 2
elif condition 3 then groupe 3
:
:
elif condition n then groupe n
else groupe n+1 fi;
```

2. Boucles

Les boucles servent à effectuer un même groupe de commandes plusieurs fois à la suite.

2.1. La boucle for

Lorsque l'utilisateur n'a pas besoin d'effectuer de test d'arrêt lors d'une boucle, par exemple dans le cas où le nombre de passages par la boucle est connu d'avance, il peut utiliser une boucle `for`. La syntaxe sous Maple est la suivante.

Boucle for (I)

```
for x from x1 to x2 by p do groupe od :
```

où x est une variable non nécessairement entière, x_1, x_2 et p sont des réels et *groupe* un ensemble d'instructions. Cette ligne de programmation effectue l'algorithme suivant : pour x variant de x_1 à x_2 avec un pas de p , exécuter les commandes *groupe*. La spécification du pas p est une option ; par défaut il vaut 1.

Le programme suivant calcule le terme d'indice 100 de la suite récurrente définie par $u_0 = 1$ et $\forall n \geq 0, u_{n+1} = \ln(1 + u_n)$.

```
> u:=1:
  for k from 1 to 100 do u:=ln(1+u): od:
  evalf(u);
```

0.01985723463

La variable de comptage (k dans l'exemple ci-dessus) n'est pas muette : elle vaut la dernière valeur calculée avant le test arrêtant la boucle.

```
> u:=1:
  for k from 1 to 100 do u:=ln(1+u): od:
  k;
```

101

Il est également possible de la laisser parcourir une liste donnée :

— Boucle for (II) —

```
for x in L do groupe od :
```

Le programme suivant crée la séquence des 5 premiers nombres pairs.

```
> L:=[0,1,2,3,4]: S:=NULL:
  for k in L do S:=S,2*k: od:
  S ;
```

0, 2, 4, 6, 8

2.2. La boucle while

La boucle **while** permet d'arrêter en cours de route les itérations dès qu'une certaine condition est vérifiée — c'est ce qu'on appelle le test d'arrêt de la boucle. La syntaxe sous Maple est la suivante.

— Boucle while —

```
while condition do groupe od ;
```

Cette ligne de programmation effectue l'algorithme suivant : tant que la *condition* est vraie, effectuer les commandes *groupe*. On prendra garde aux boucles infinies, c'est-à-dire celles qui ne se terminent jamais parce que la condition est toujours vérifiée au cours des itérations ! Dans le cas d'une « boucle folle », on tentera un arrêt grâce à l'onglet **stop** de la barre d'outil. Le programme suivant calcule la séquence des cubes d'entiers inférieurs à 50.

```
> k:=1: S:=NULL: while k^3<=50 do S:=S,k^3: k:=k+1: od;
```

```
S := 1
```

```
k := 2
```

```
S := 1, 8
```

```
k := 3
```

```
S := 1, 8, 27
```

```
k := 4
```

Ajoutons que l'utilisateur peut empêcher l'affichage des calculs itérés de la boucle en remplaçant `od ;` par `od : S ;`. Il faudra alors forcer le logiciel à afficher le résultat *après* la boucle.

```
> k:=1: S:=NULL: while k^3<=50 do S:=S,k^3: k:=k+1: od: S;
```

```
1, 8, 27
```

Une boucle permet par exemple de calculer la valeur d'une somme ou d'un produit. Cette structure est également adaptée au calcul des termes d'une suite récurrente. Le programme suivant calcule la somme des 100 premiers entiers naturels¹.

```
> somme:=0: k:=0: while k<=99 do somme:=somme+k: k:=k+1: od: somme;
```

```
4950
```

Précisons que l'on peut combiner les deux types de boucles `for` et `while`. La syntaxe est la suivante :

— Boucle `for` avec test d'arrêt —

```
for x from x1 to x2 by p while condition do groupe od :
```

2.3. Exemple : calcul d'une valeur approchée

Soit f la fonction définie sur \mathbb{R} par $f(x) = x^7 + x + 1$. Cette fonction est dérivable sur \mathbb{R} et

$$\forall x \in \mathbb{R}, f'(x) = 7x^6 + 1 > 0.$$

De plus

$$\lim_{x \rightarrow -\infty} f(x) = -\infty \text{ et } \lim_{x \rightarrow +\infty} f(x) = +\infty.$$

La fonction f réalise donc une bijection strictement croissante de \mathbb{R} sur \mathbb{R} . L'équation $f(x) = 0$ admet par conséquent une unique solution α . Le programme suivant calcule une valeur approchée de α à 10^{-3} près par *dichotomie*².

1. Signalons que la commande `add` effectue plus rapidement ce calcul : la ligne de commandes `add(i,i=0..99)` ; renvoie le résultat 4950.

2. cf. le cours d'analyse et en particulier la preuve du théorème des valeurs intermédiaires.

```

> f:=x->x^7+x+1: Digits:=4: a:=-1: b:=1:
  while evalf(b-a)>0.0001 do
    if evalf(f((a+b)/2))<0 then a:=(a+b)/2:
    else b:=(a+b)/2:
    fi:
  od:
a;

```

-0.7966

3. 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: fi: Max;

```

2

L'inconvénient d'un tel programme est qu'il faut modifier l'affectation de x et y avant de lancer le corps du programme... L'idéal serait utile de disposer d'une écriture semblable à celle des fonctions : définir une fonction Max renvoyant $\text{Max}(x, y)$ sans avoir à écrire systématiquement le programme ou les affectations de x et y en utilisant le copier-coller. C'est précisément l'objet des procédures sous Maple.

3.1. 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 proc:

```

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

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

```
12.2231
```

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 k,s; s:=0; for k from 1 to 1000
do s:=s+k^2: od: end proc:
> 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 qui calcule la somme $1^2 + 2^2 + \dots + 1000^2$.

```
> f:=proc() local k,s; s:=0; for k from 1 to 1000
do s:=s+k^2: od: end proc:
> s:=5: f(),s;
```

```
333833500,5
```

```
> g:=proc() local k; global s; s:=0: for k from 1 to 1000
do s:=s+k^2: od: end proc:
> s:=5: g(),s;
```

```
333833500,333833500
```

Le lecteur retiendra la syntaxe suivante pour une procédure en fonction des arguments a_1, \dots, a_n .

Syntaxe pour la création d'une procédures

```
nom := proc ( $a_1, \dots, a_n$ )
local  $\ell_1, \dots, \ell_m$  ; global  $g_1, \dots, g_r$  ;
Corps du programme
end proc :
```

La ligne de commande $\mathbf{nom}(a_1, \dots, a_n)$ renvoie le dernier résultat calculé par la procédure « \mathbf{nom} ». On prendra garde à ce que Maple traite les arguments a_k 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 a_k . On utilise dans ce cas des variables locales.

3.2. Procédures récursives

Il est même possible de programmer une procédure qui « s'appelle lui-même » au cours de son exécution ! Ce type de procédure s'appelle procédure récursive et est adapté aux objets mathématiques définis par récurrence. On utilise la commande `RETURN` pour forcer la valeur de sortie de la procédure. Voici l'exemple d'une procédure calculant $n!$.

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

26525285981219105863630848000000
```

Le fonctionnement de cette procédure est simple. L'utilisateur écrit la commande `fact(30)` ; le logiciel doit alors calculer $30 \times \text{fact}(29)$, il « lance » donc la procédure pour la valeur 29 ; il doit alors calculer $29 \times \text{fact}(28)$, 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

$$\mathbf{fact(30)} := 30 \times \mathbf{fact(29)} := 30 \times 29 \times \mathbf{fact(28)} := \dots := 30 \times 29 \times \dots \times \mathbf{fact0} := 30 \times 29 \times \dots \times 1$$

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'exécution du programme en conservant en mémoire la trace de ses calculs. (Mais la vitesse dépend aussi de la version de Maple ainsi que du matériel utilisé.)

```
> time(fact(20000));

0.609
> factremember := proc(n) option remember;
      if n=0 then RETURN(1)
      else RETURN(n*factremember(n-1))
      fi: end proc:
> time(factremember(20000));

0.547
```

4. Exercices

Exercice 1.

Executer les deux programmes suivants. Pourquoi donnent-ils des résultats différents ?

```
> n:=1 : S:=0 : while S<10 do S:=S+n : n:=n+1 : od:
n; S;

> n:=1 : while sum(k,k=1..n)<10 do n:=n+1 : od:
n; sum(k,k=1..n);
```

Exercice 2.

Ecrire une boucle affichant tous les nombres premiers compris entre 1 et 1000. On utilisera `isprime`.

Exercice 3.

Déterminer les couples d'entiers naturels consécutifs (a, b) avec $0 \leq a, b \leq 100$ tels que $ab + 1$ soit le cube d'un entier.

Exercice 4.

On pose

$$\forall n \in \mathbb{N}^*, H_n = \sum_{k=1}^n \frac{1}{k}.$$

On admet que H_n tend en croissant vers $+\infty$ quand n tend vers $+\infty$. Déterminer le plus petit entier n tel que $H_n > 7$.

Exercice 5.

Vous prouverez dans le cours d'analyse réelle que la suite de terme général

$$u_n = \sum_{k=0}^n \frac{1}{k!}$$

converge vers e et que $\forall n \geq 0, |e - u_n| \leq \frac{3}{(n+1)!}$. En déduire un programme calculant une valeur approchée de e à 10^{-8} près.

Exercice 6.

Ecrire un programme calculant par dichotomie à 10^{-6} près l'unique racine sur $[1, +\infty[$ de $1 + 8x - x^8 = 0$.

Exercice 7.

La suite de Fibonacci est définie par

$$u_0 = 1, u_1 = 1, \forall n \geq 0, u_{n+2} = u_{n+1} + u_n.$$

Ecrire un programme renvoyant la valeur de u_{1000} .

Exercice 8.

Ecrire, sans utiliser la commande `solve`, une procédure `Sol` pour résoudre l'équation $ax^2 + bx + c = 0$, $x \in \mathbb{R}$, pour tout $(a, b, c) \in \mathbb{R}^3$.

Exercice 9.

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 de la suite définie par

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

Exercice 10.

Pour $(n, m) \in \mathbb{N}^2$ on note $S(n, m)$ le nombre de surjections de $\llbracket 1, n \rrbracket$ sur $\llbracket 1, m \rrbracket$.

1. Que vaut $S(n, n)$ pour $n \in \mathbb{N}^*$? Que vaut $S(n, m)$ si $n < m$?
2. Que vaut $S(0, 0)$? Et $S(n, 0)$ pour $n \in \mathbb{N}^*$?
3. Montrer que pour tout $(n, m) \in \mathbb{N}^2$, $S(n + 1, m) = m(S(n, m) + S(n, m - 1))$.
4. Ecrire une procédure sous Maple pour la fonction $S : \mathbb{N}^2 \rightarrow \mathbb{N}$.

Exercice 11.

Ecrire un procédure qui donne le quotient et le reste de la division euclidienne de $n \in \mathbb{N}$ par $m \in \mathbb{N}^*$.

Exercice 12.

Aujourd'hui est lundi 1er janvier 2018 et il est 0h. Ecrire une procédure qui affiche le jour de la semaine et l'heure qu'on sera dans n heures (avec $n \in \llbracket 0, 8760 \rrbracket$).

Exercice 13.

Ecrire un procédure qui donne la décomposition d'un nombre naturel en facteurs premiers. (Le but de l'exercice est de ne pas utiliser la commande `ifactor` ; en revanche vous pouvez faire appel aux équivalences avec `mod`).

Exercice 14.

En 1742 le mathématicien allemand Christian Goldbach (1690-1764) écrivit une lettre au mathématicien suisse Leonhard Euler dans laquelle il proposait la conjecture suivante :

Tout nombre entier pair strictement supérieur à 2 peut être écrit comme la somme de deux nombres premiers (éventuellement deux fois le même).

1. Ecrire une procédure `Goldbach` qui à tout entier pair $n \geq 4$ renvoie deux nombres premiers de somme n s'il en existe ou un message d'erreur dans le cas contraire.
2. Tester la conjecture de Goldbach pour des entiers entre 4 et 1000.

A ce jour la conjecture reste ouverte... En 2008 elle a été vérifiée par ordinateur pour tous les nombres pairs jusqu'à $1,1 \times 10^{18}$. En 2000, afin de faire de la publicité pour le livre *Uncle Petros and Goldbach's Conjecture* de Apostolos Doxiadis, un éditeur britannique offrit un prix de 1 000 000 \$ pour une preuve de la conjecture. Le prix ne pouvait être attribué qu'à la seule condition que la preuve soit soumise à la publication avant avril 2002. Il n'a jamais été réclamé.