# TP Maple 3 | Tests et boucles

*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.*



## 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 :

```
> 2=3: evalb(%);
```

false

On prendra également garde aux effets d'arrondis, comme en témoigne l'exemple suivant...

```
> 5>4.99999: evalb(%);
```

true

```
> 5>4.99999999999999999999999999999999: evalb(%);
```

false

Enfin, l'utilisateur se souviendra que les expressions construites au moyen des opérateurs de comparaison (<, >, etc.) ne peuvent être évaluées par Maple que si elles portent sur des objets de type *numeric*.

```
> evalb(sqrt(3)<1);
```

$1 > \sqrt{3}$

L'utilisateur dispose également d'opérateurs logiques permettant de construire des expressions booléennes plus élaborées : il s'agit des opérateurs **not** (le contraire), **and** (et) et **or** (ou).

Opérations sur les booléens

- **not** : le contraire.
- **and** : la conjonction « et ».
- **or** : le « ou » non disjonctif.

```
> (2<1) or (4<5), (2<1) and (4<5), not(1<-1);
```

*true, false, true*

### 1.2. La structure de test if ... then

Un test permet d'effectuer des groupes de commandes si certaines conditions sont vérifiées. La syntaxe est la suivante.

Syntaxe d'un test ( I )

**if** *condition* **then** *commandes à effectuer* **fi;**

où « *condition* » est une condition (expression booléenne) et « *commandes à effectuer* » un groupe d'instructions à effectuer si la *condition* est vérifiée.

Par exemple :

```
> a:=2:if (a<3) then print(‘a est strictement inférieur à 3‘) fi;
```

a est strictement inférieur à 3

On peut aussi utiliser la syntaxe suivante.

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:=2:if (a<3) then print(‘a est strictement inférieur à 3‘) else print(‘a est
  supérieur ou égal à 3‘) fi;
```

a est strictement inférieur à 3

```
> a:=4:if (a<3) then print(‘a est strictement inférieur à 3‘) else print(‘a est
  supérieur ou égal à 3‘) fi;
```

a est supérieur ou égal à 3

Dans le cas où il y a plus de deux conditions, on utilisera la syntaxe indiquée ci-dessous.

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 ;**

Le lecteur l'aura compris, la commande **elif** est une contraction de else-if ; on l'utilise dans le cas d'une alternative entre plus de trois termes. Voici, en guise d'exemple, un programme testant si un nombre réel donné est strictement plus grand que 2.

```
> x:=0; if (x>2) then print(‘test positif‘) else  print(‘test négatif‘) fi;
```

test négatif

```
> x:=6; if (x>2) then print(‘test positif‘) else  print(‘test négatif‘) fi;
```

test positif

Comme nous l'avons remarqué dans le paragraphe sur les expressions booléennes, les opérateurs de comparaison doivent porter sur des objets de type *numeric* sous peine d'un message d'erreur.

```
> a:=sqrt(2): if (a<1) then print(‘a<1‘) else print(‘a>=1‘) fi;
Error, cannot determine if this expression is true or false: 2^(1/2) < 1
```

On pourra contourner cette difficulté au moyen de la commande **is**...

```
> a:=sqrt(2): if is(a<1) then print(‘a<1‘) else print(‘a>=1‘) fi;
```

$a >= 1$

ou encore grâce à **evalf**.

```
> a:=sqrt(2): if (evalf(a)<1) then print(‘a<1‘) else print(‘a>=1‘) fi;
```

$a >= 1$

## 2. Boucles

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

### 2.1. 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 100.

```
> k:=1:
  S:=NULL:
  while (k^3<=100) do S:=S,k^3: k:=k+1: od;
```

$$S := 1$$
$$k := 2$$
$$S := 1,8$$
$$k := 3$$
$$S := 1,8,27$$
$$k := 4$$
$$S := 1,8,27,64$$
$$k := 5$$
$$L := 1,8,27,64$$

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

```
> k:=1: S:=NULL:
  while (k^3<=100) do S:=S,k^3: k:=k+1: od: S;
```

$$1,8,27,64$$

Une boucle permet par exemple de calculer la valeur d'une somme ou d'un produit. Cette structure est également adpatée au calcul des termes d'une suite récurrente. Le programme suivant calcule la somme des 100 premiers entiers naturels [1].

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

$$4950$$

### 2.2. 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 passage 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** $x_1$ **to** $x_2$ **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*. Il est également possible de faire varier $x$ dans une liste, grâce au codage suivant.

Boucle **for** (II)

**for** $x$ **in** $L$ **do** *groupe* **od :**

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

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

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

$$0$$

$$0,2$$

$$0,2,4$$

$$0,2,4,6$$

$$0,2,4,6,8$$

Ajoutons que l'utilisateur peut empêcher l'affichage des calculs itérés de la boucle en remplaçant **od;** par **od :**, il faudra alors forcer Maple à afficher le résulat après la boucle. Le programme suivant calcule le terme d'indice 100 de la suite récurrente définie par $u_0 = 1$ et $\forall\, n \geqslant 0$, $u_{n+1} = \ell\mathrm{n}(1+u_n)$.

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

$$0.01985723463$$

La variable de comptage ($k$ dans les exemples 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;
```

$$101$$

Précisons que l'on peut utiliser un test d' arrêt dans une boucle **for**. La syntaxe est la suivante :

Boucle **for** avec test d'arrêt

**for** $x$ **from** $x_1$ **to** $x_2$ **by** $p$ **while** *condition* **do** *groupe* **od :**

Le logiciel arrêtera automatiquement la boucle dès la *condition* ne sera plus vérifiée. On pourra aussi utiliser judicieusement la commande **break** au sein d'une boucle pour forcer son arrêt.

Commande **break**

La commande **break** permet d'arrêter automatiquement une boucle **for** ou **while**.

## 3. Un exemple plus élaboré

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

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

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 $\alpha$. Le programme suivant calcule une valeur approchée de $\alpha$ à $10^{-3}$ près par *dichotomie*[2].

```
> 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;
```

$$-0.7964$$

## 4. Exercices

**Exercice 1.**

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

**Exercice 2.**

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

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

**Exercice 3.**

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 $H_n > 100$.

**Exercice 4.**

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

**Exercice 5.**

Pour tout entier relatif $n$, on sait qu'il existe un unique polynôme $T_n$ réel tel que

$$\forall t \in \mathbb{R},\ \cos(nt) = T_n(\cos(t)).$$

$T_n$ est appelé le $n$-ième polynôme de Tchebychev de seconde espèce. On se propose de calculer les valeurs de $T_n$ pour $n$ variant de 0 à 20 au moyen de boucles de trois manières différentes...

**1**. Rechercher dans l'aide en ligne la commande permettant de calculer $T_n$. Ecrire une boucle affichant les valeurs de $T_n$ pour $n$ variant de 0 à 20.

**2**. Sans utiliser la commande prédéfinie par le logiciel, en revenant simplement à la définition donnée ci-dessus de $T_n$, écrire une boucle calculant les valeurs de $T_n$ pour $n$ variant de 0 à 20. On utilisera la commande **expand**.

**3**. On prouve que la suite $(T_n)_{n\geqslant 0}$ vérifie la relation de récurrence suivante :

$$T_0 = 1,\ \ T_1 = X,\ \ \forall n \geqslant 0,\ \ T_{n+2} = 2XT_{n+1} - T_n.$$

En déduire une boucle calculant les valeurs de $T_n$ pour $n$ variant de 0 à 20.

**Exercice 6.**

La suite de Fibonacci est définie par

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

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

**Exercice 7.**

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