Programmer avec le logiciel Maple (1)

I Introduction

I.1 Principe de la programmation

De nombreuses méthodes mathématiques sont de simples enchaînements de calculs prédéfinis. Il serait très fastidieux de devoir les adapter systématiquement aux cas particuliers que nous rencontrons. De plus, il est parfois humainement impossible de faire certains calculs ; ne serait-ce que pour le temps qu'ils nécessitent. Pour éviter cela, il faudrait pouvoir automatiser l'exécution de ces calculs en indiquant à Maple la marche à suivre pour réaliser la méthode et ce d'une manière aussi indépendante du contexte que possible.

I.2 La notion d'algorithme

La notion d'algorithme généralise cette idée. En voici quelques définitions :

1. "suite d'opérations selon un processus défini aboutissant à la résolution d'un problème"
2. "processus logique permettant la résolution d'un problème en programmation"
3. "Méthode de calcul qui indique la démarche à suivre pour résoudre une série de problèmes équivalents en appliquant dans un ordre précis une suite finie de règles."

Par exemple, la résolution dans $\mathbb{R}$ des équations du second degré à coefficients réels de la forme $ax^2+bx+c=0$ se résume aux opérations suivantes :

1. Calculer $\Delta = b^2 - 4ac$.
2. On a alors trois cas :
   (a) Si $\Delta > 0$, alors les solutions de l'équation sont : $\dfrac{-b+\sqrt{\Delta}}{2a}$ et $\dfrac{-b-\sqrt{\Delta}}{2a}$.
   (b) Si $\Delta = 0$, alors l'unique solution de l'équation est : $-\dfrac{b}{2a}$.
   (c) Si $\Delta < 0$, alors il n'y a pas de solution (réel).

Nous venons d'écrire notre premier algorithme ! Il nous reste à le mettre en oeuvre.

Ce l'on appelle couramment la programmation est en fait la mise en oeuvre d'algorithmes. Pour cela, le logiciel Maple nous fournit, en plus de ses fonctionnalités mathématiques, un langage de programmation ; c'est à dire un jeu d'instructions spécifiques. La programmation sous Maple consiste ainsi à traduire nos algorithmes dans ce langage.

Parmi les instructions de programmation, nous avons par exemple :

1. La commande "`if...then...else...fi ;`" permet de tester la valeur de variables. Par exemple, dans l'exemple précédent, après le calcul de Δ, il est nécessaire de tester sa valeur pour pouvoir calculer les solutions de l'équation du second degré.
2. La commande "`for...do...od ;`" permet de répéter une série d'opérations un nombre prédéfini de fois.
3. La commande "`while...do...od ;`" permet de répéter une série d'opérations tant qu'une certaine condition est vérifiée.

Nous détaillerons le fonctionnement de ces commandes lors des prochaines séances.

I.3 Principes d'écriture d'un algorithme

Le logiciel Maple ne comprend pas ce que vous faites. En particulier, il ne comprend pas le problème que vous chercher à résoudre. Il ne fait qu'enchaîner les calculs que vous lui demander de faire. Il ne réfléchit pas. (D'ailleurs, c'est pour cela qu'il est si efficace.)

Pour pouvoir résoudre le problème que vous vous posez, il faut transformer sa résolution en un algorithme ; c'est à dire en une suite d'opérations aussi élémentaires que possible. Enfin, il faut l'écrire dans le langage Maple ; c'est à dire traduire cette suite d'opérations en une suite d'instructions Maple.

Pour pouvoir créer et mettre en oeuvre un algorithme, il faudra respecter plusieurs étapes indispensables :

1. Identifier les paramètres de votre problème.
2. Identifier **de manière très détaillée** l'enchaînement **chronologique** des calculs à effectuer.
3. Identifier les données qui devront être stockées dans des variables.
4. Rédiger l'algorithme proprement dit dans un langage simple facilement traduisible en celui de Maple. Par exemple, nous représenterons les affectations par l'écriture de Maple : `var :=valeur`.
5. Traduire en langage Maple cet algorithme.

Exemple I.1. *Comment programmer la résolution des équations du second degré ?*

1. *Paramètres du problème : les coefficients a, b et c.*
2. *Enchaînement des calculs :*
   (a) *Calculer le discriminant Δ.*
   (b) *Tester la valeur de Δ et renvoyer la solution de l'équation en fonction de la valeur de Δ (trois cas)*
3. *Données à stocker : Δ*
4. *Nous réécrivons l'algorithme précédent de manière plus précise et, nous le verrons, plus proche du langage de Maple :*

   Stocker les valeurs de a, b et c.
   $\Delta := b^2 - 4ac$
   Si $\Delta > 0$ alors :
   Calculer : $\{\frac{-b+\sqrt{\Delta}}{2a}, \frac{-b-\sqrt{\Delta}}{2a}\}$
   Sinon,
   Si $\Delta = 0$ alors :
   Calculer : $\{-\frac{b}{2a}\}$
   Sinon :
   Calculer : $\emptyset$ *(noté :* { } *sous Maple)*
   Fin si
   Fin si

5. *Nous verrons dans ce TP les notions nécessaires pour pouvoir écrire l'algorithme proposé.*

II Boucles conditionnelles

Lors de la mise en oeuvre d'un algorithme, on a souvent besoin de tester des propriétés pour pouvoir altérer l'enchaînement des calculs. Par exemple, lors de la résolution dans $\mathbb{R}$ d'une équation du second degré à coefficients réels, on procède différemment selon le signe du discriminant. Bien d'autres méthodes se présentent sous la forme d'une alternative en fonction d'une proposition *ad hoc*. Les structures conditionnelles permettent de traiter ce genre de situations.

II.1 Booléens

Un booléen est une variable à deux états : vrai ou faux (true ou false sous Maple). La fonction `evalb` en permet l'évaluation.

Exemple II.1.

```
> evalb(x<0) ;
```

$$x < 0$$

Ici, la variable `x` ne contient aucune valeur. Maple nous renvoie l'inéquation.

```
> x :=3 ;
  x<0 ;
  evalb(x<0) ;
```

$$x := 3$$
$$3 < 0$$
$$false$$

Remarquons que Maple fait l'évaluation booléenne de l'inéquation que si nous le lui demandons.

Remarque II.1. Rappel : *Pour représenter les inégalités, Maple utilise la syntaxe suivante :*

1. $\leq$: `<=`
2. $\geq$: `>=`
3. $\neq$: `<>`

Il est possible de relier les booléens par les opérateurs logiques de conjonction (et exclusif), disjonction (ou inclusif) et de négation (non). Rappelons leurs tables de vérité :

Table de vérité de la loi "et"

a	b	a et b
F	F	F
F	V	F
V	F	F
V	V	V

Table de vérité de la loi "ou"

a	b	a ou b
F	F	F
F	V	V
V	F	V
V	V	V

Table de vérité de la loi "non"

a	non(a)
F	V
V	F

Sous Maple, le nom de ces opérateurs est traduit en anglais. Plus précisément, on a les syntaxes suivantes :

`a and b`	proposition `a` et `b`
`a or b`	proposition `a` ou `b`
`not(a)`	négation de la proposition `a`

Exemple II.2.

```
> x :=3 ;
  (x>0) and (x<1) ;
  (x>0) or (x<1) ;
  not(x>0) ;
```

$$x := 3$$
$$false$$
$$true$$
$$false$$

Remarque II.2. *Maple refuse de tester des inégalités avec des expressions non numériques. Par exemple, demandons à Maple si $\pi > 0$.*

```
> evalb(Pi>0) ;
```

$$0 < \pi$$

Maple n'est simplement pas capable de faire un tel test. Une telle proposition sera donc inexploitable lorsque l'on programmera. Pour palier à cela, il nous faudra penser à faire une évaluation numérique des termes que l'on exploite grâce à la fonction `evalf` *:*

```
> evalb(evalf(Pi>0)) ;
```

$$true$$

II.2 if...then...else...

Comme nous l'avons déjà expliqué, il est souvent utile d'altérer un enchaînement de calculs en fonction des résultats intermédiaires trouvés. Pour cela, on utilise des boucles `if`. La syntaxe minimale de ces boucles est la suivante :

```
> if condition then
        instructions ;
  fi ;
```

Avec cette commande, Maple n'exécutera les `instructions` que si la `condition` est vraie.

Exemple II.3.

```
> x :=3 :
  if x>0 then
      x^2 ;
  fi ;
```

```
> x :=-3 :
  if x>0 then
      x^2 ;
  fi ;
```

Remarque II.3. *On n'oubliera pas de marquer la fin de la boucle* `if` *par la commande* `fi`*. Elle sert à indiquer quelles sont les instructions qui doivent être exécuter sous la condition demandée et celles qui doivent exécuter après la boucle* `if`*. En effet, Maple exécutera les instructions situées après la commande* `fi` *quel que soit le résultat du test de la condition.*

Dans le cas où la `condition` est fausse, il peut être nécessaire d'exécuter d'autres instructions. Pour cela, la syntaxe est la suivante :

```
> if condition then
          instructions_1 ;
  else
          instructions_2 ;
  fi ;
```

Avec cette instructions, Maple exécute les instructions $\texttt{instructions}_1$ si la condition `condition` est vraie. Sinon, elle exécute les instructions $\texttt{instructions}_2$. Ces dernières instructions sont exécutées si, et seulement si, la condition `condition` est fausse.

Exemple II.4.

```
> x :=3 :
  if x>0 then
      x^2 ;
  else
      -x^2 ;
  fi ;
```

$$9$$

```
> x :=-3 :
  if x>0 then
      x^2 ;
  else
      -x^2 ;
  fi ;
```

$$-9$$

Dans le cas où la `condition` n'est pas vraie, on peut également tester une seconde contition. La syntaxe est alors la suivante :

```
> if condition_1 then
          instructions_1 ;
  elif condition_2 then
          instructions_2 ;
  fi ;
```

Maple exécute les instructions $\texttt{instructions}_1$ si la condition $\texttt{condition}_1$ est vérifiée. **Sinon**, il teste la condition $\texttt{condition}_2$. Si celle-ci est vraie, il exécute les instructions $\texttt{instructions}_2$. Ces instructions sont donc exécutées si, et seulement si la condition $\texttt{condition}_1$ est fausse et la condition $\texttt{condition}_2$ est vraie.

Exemple II.5.

```
> x :=3 :
  if x>0 then
      x^2 ;
  elif x>-1 then
      -x^2 ;
  fi ;
```

$$9$$

```
> x :=-1 :
  if x>0 then
      x^2 ;
  elif x>-1 then
      -x^2 ;
  fi ;
```

$$-1$$

```
> x :=-3 :
  if x>0 then
      x^2 ;
  elif x>-1 then
      -x^2 ;
  fi ;
```

On peut répéter les commandes `elif` autant de fois que nécessaire et compléter cela avec la commande `else`. La syntaxe générale d'une boucle `if` est alors :

```
> if conditions_1 then
          instructions_1 ;
  elif conditions_2 then
          instructions_2 ;
  ⋮
  elif conditions_{n-1} then
          instructions_{n-1} ;
  else
          instructions_n ;
  fi ;
```

où :

1. `conditions`$_1$, `conditions`$_2$, ..., `conditions`$_{n-1}$ désignent des booléens exprimant des conditions (sic).
2. `instructions`$_1$, `instructions`$_2$, ..., `instructions`$_n$ désignent des suites d'instructions.
3. `if` signifie "si".
4. `then` signifie "alors".
5. `else` signifie "sinon".
6. `elif` signifie "sinon si" ; `elif` est la contraction de `else if`.
7. `fi` marque la fin de la boucle `if`.

L'exécution d'une telle commande se fait de la façon suivante : Si la condition `conditions`$_1$ est vraie, alors on exécute les instructions `instructions`$_1$. **Sinon**, si la condition `conditions`$_2$ est vraie, alors on exécute les instructions `instructions`$_2$, *etc...*, **sinon** (si aucune des conditions précédentes n'est vérifiée) on exécute les instructions `instructions`$_n$.

Exemple II.6.

```
> x :=3 :
  if x>0 then
      x^2 ;
  elif x>-1 then
      -x^2 ;
  else
      x^3 ;
  fi ;
```

9

```
> x :=-1 :
  if x>0 then
      x^2 ;
  elif x>-1 then
      -x^2 ;
  else
      x^3 ;
  fi ;
```

−1

```
> x :=-3 :
  if x>0 then
      x^2 ;
  elif x>-1 then
      -x^2 ;
  else
      x^3 ;
  fi ;
```
$$-27$$

Remarque II.4. *On fera particulièrement attention à bien présenter ses boucles de telle sorte à les rendre aussi lisibles que possible. Par exemple, comme ci-dessus, on pourra aligner les instructions et les décaler de quelques espaces (3 à 5) par rapport aux commandes* `if, then, elif, else, fi`.

Exercice 1. *Valeur absolue*

Créer une suite d'instructions calculant la valeur absolue de `x` en revenant précisément à la définition de la fonction valeur absolue. On testera les cas : $x = 3$, $x = \pi$, $x = -1$, $x = -\sqrt{2}$.

Exercice 2. *Équation du second degré*

Calculer les solutions de l'équations $ax^2 + bx + c = 0$ en vous appuyant sur l'algorithme proposé précédemment. On traitera les cas : $(a, b, c) = (1, 1, -1)$, $(a, b, c) = (1, 6, 9)$, $(a, b, c) = (1, 1, 1)$ et $(a, b, c) = (1, 1, -\sqrt{2})$.

Exercice 3. *Fonction définie par morceaux*

On considère la fonction f :
$$\begin{cases} \mathbb{R} & \longrightarrow & \mathbb{R} & \\ x & \longmapsto & \tan^2(\frac{\pi}{4}\sin(x)) - \tan(\frac{\pi}{4}\sin(x)) & \text{si } x < 0 \\ x & \longmapsto & \tan(\frac{\pi}{4}\sin(x)) & \text{si } 0 \leq x < \frac{\pi}{2} \\ x & \longmapsto & \tan^2(\frac{\pi}{4}\sin(x)) - \tan(\frac{\pi}{4}\sin(x)) + 1 & \text{si } x \geq \frac{\pi}{2} \leq x < \pi \\ x & \longmapsto & -\tan(\frac{\pi}{4}\sin(x)) + 1 & \text{si } x \geq \pi \end{cases}$$
Calculer $f(x)$ pour $x = -\frac{\pi}{4}$, $x = \frac{\pi}{4}$, $x = \frac{3\pi}{4}$ et $x = 2\pi$.
Consigne : On proposera une suite d'instructions aussi courte que possible.

III Boucles itératives

Jusqu'à présent, nous avons pu enchaîner des instructions et modifier leur enchaînement en fonction de résultats intermédiaires grâce aux boucles `if`. A ce stade, nous ne pouvons cependant pas enchaîner des instructions de manière répétitive. Par exemple, nous ne pouvons calculer $n!$ pour tout n sans utiliser la commande spécifique de Maple. Prenons le cas $n = 6$. On considère alors la formule suivante : $6! = 1 \times 2 \times 3 \times 4 \times 5 \times 6$ et nous enchaînons les calculs suivants :

1. $1 \times 2 = 2$
2. $2 \times 3 = 6$
3. $6 \times 4 = 24$
4. $24 \times 5 = 120$
5. $120 \times 6 = 720$

Dans ce cas de figure, nous sommes bien contraints d'enchaîner des calculs de manière répétitive. En programmation, on parle de boucle itérative. Nous verrons deux types de boucles itératives : les boucles `for` et les boucles `while` (au T.P. suivant).

Remarque III.1. *La gestion des variables est dynamique ; c'est à dire que la donnée stockée dans une variable peut évoluer dans le temps. On se servira de cette propriété pour conserver temporairement dans une seule variable les résultats de plusieurs calculs intermédiaires. En effet, nous ne pouvons créer qu'un nombre prédéterminé de variables. Il faudra donc repérer sur quelle période de temps nous avons besoin de conserver chaque donnée. Cela est indispensable lorsque l'on répète un grand nombre de fois des calculs.*
Dans l'exemple précédent, il est inutile de conserver tous les calculs intermédiaires. Il suffit de stocker successivement dans **une seule variable** *1, 2, 6, 24, 120 et 720. Il faut cependant savoir à quelle étape du calcul nous sommes. Cela sera stocker dans une seconde variable.*

III.1 Boucles `for`

Une première façon d'enchaîner des instructions de manière répétitive est d'itérer une suite d'instructions un nombre de fois prescrit. Sous Maple, comme d'ailleurs pour de nombreux langages de programmation, on utilise pour cela une boucle `for`. La syntaxe de ces boucles est la suivante :

```
> for var from ini to fin do
      Instruction_1 ;
      Instruction_2 ;
      ...
  od ;
```

où :

1. `var` est le nom d'une variable. On l'appellera variable d'itération de la boucle `for`.
2. `ini` indique la valeur d'initialisation de la boucle `for`.
3. `fin` indique la valeur de fin de la boucle `for`.
4. $\texttt{Instruction}_1$, $\texttt{Instruction}_2$, ... sont des instructions Maple dépendant éventuellement de `var`.
5. `od` indique la fin de la boucle `for`.

Une telle instruction fait varier la variable `var` de la valeur `ini` jusqu'à la valeur `fin`. A chaque étape, la variable `var` est augmentée d'une unité et les instructions $\texttt{Instruction}_1$, $\texttt{Instruction}_2$, ... sont exécutées.

Remarque III.2. *Une boucle `for` sert à enchaîner une suite de calculs identiques lors d'un nombre prédéfini d'étapes.*

Remarque III.3. *Il est très important d'avoir en tête le contenu des différentes variables que l'on souhaite avoir à la fin de chaque itération de la boucle.*

Exemple III.1. *Comment programmer le calcul de la somme* $\sum_{k=1}^{5} k$ *sans utiliser les fonctions `sum` et `add` de Maple ?*
Nous suivons la méthodologie proposée au début du TP :

1. *Ici, il n'y a pas de paramètre au problème.*
2. *Nous devons enchaîner les calculs suivants :*
   (a) 0
   (b) $0 + 1 = 1$
   (c) $1 + 2 = 3$
   (d) $3 + 3 = 6$
   (e) $6 + 4 = 10$
   (f) $10 + 5 = 15$
3. *Nous stockerons ces calculs intermédiaires dans une variable `S`. De plus, implicitement, cette suite d'opérations utilise la variable k servant à définir la somme. En effet, elle indique à quelle étape du calcul nous sommes. Il nous faut donc une variable `k` que l'on va faire varier de* 1 *à* 5 *grâce à une boucle `for`.*
4. *Il nous faut définir un algorithme.*
   (a) *Au départ, nous partons de* 0. *Nous stockons cela dans `S`.*
   (b) *Puis, nous y ajoutons successivement* 1, 2, 3, 4 *et enfin :* 5.
   Nous faisons varier la variable `k` de 1 *à* 5. *A chaque étépe, nous ajoutons `k` à la variable `S`.*

   Nous avons donc l'algorithme suivant :
   (a) $S := 0$ *(initialisation)*
   (b) *pour k variant de 1 à 5 :*
   $S := S + k$ *(à la fin de chaque étape, $S = 1 + 2 + \cdots + k$)*
   fin pour

5. *La suite d'instructions suivante calcule* $\sum_{k=1}^{5} k$.

```
> S :=0 ;
  for k from 1 to 5 do
       S :=S+k ;
  od ;
  S ;
```

$$S := 0$$
$$S := 1$$
$$S := 3$$
$$S := 6$$
$$S := 10$$
$$S := 15$$
$$15$$

Remarque III.4. *Pour éviter l'affichage des calculs intermédiaires, on mettra " :" derrière* `od` *si l'on ne souhaite pas connaître les résultats des calculs intermédiaires. Le fait d'en mettre derrière les instructions de la boucle sera sans effet.*

On peut spécifier un pas d'itération pour la boucle `for`. La syntaxe est alors la suivante :

```
> for var from ini to fin by pas do
        Instruction1 ;
        Instruction2 ;
        ...
  od;
```

où : `pas` désigne le pas d'itération.
Une telle boucle fait varier la variable `var` en augmentant à chaque étape de `pas` depuis `ini` jusqu'à `fin`. A chaque itération, les instructions $\texttt{Instruction}_1$, $\texttt{Instruction}_2$, ... sont exécutées.

Remarque III.5. *Il est tout à fait possible de spécifier un pas négatif.*

Exemple III.2. *Calculons :* $\sum_{\substack{k=1 \\ k \ impair}}^{11} \frac{1}{k^2}$.

```
> S :=0 ;
  for k from 1 to 11 by 2 do
      S :=S+1/k^2 ;
  od ;
  S ;
```

$$S := 0$$
$$S := 1$$
$$S := \frac{10}{9}$$
$$S := \frac{259}{225}$$
$$S := \frac{12916}{11025}$$
$$S := \frac{117469}{99225}$$
$$S := \frac{14312974}{12006225}$$
$$\frac{14312974}{12006225}$$

Les commandes suivantes sont parfois utiles.

`break`	sort de la boucle.
`next`	passe à l'occurence suivante de la boucle.

Exemple III.3. *La commande* `type(n,even)` *répond vrai si* `n` *est un nombre pair et faux sinon. Grâce à l'instruction* `next`*, nous pouvons réécrire la suite d'instruction précédente de la façon suivante :*

```
> S :=0 ;
  for k from 1 to 11 do
      if type(k,even) then
          next ;
      fi ;
      S :=S+1/k^2 ;
  od ;
  S ;
```

$$S := 0$$
$$S := 1$$
$$S := \frac{10}{9}$$
$$S := \frac{259}{225}$$
$$S := \frac{12916}{11025}$$
$$S := \frac{117469}{99225}$$
$$S := \frac{14312974}{12006225}$$
$$\frac{14312974}{12006225}$$

Exemple III.4. *Grâce à la fonction* `break`*, nous pouvons arrêter notre calcul dès que nous dépassons la valeur* $1,15$ *(par exemple) :*

```
> S :=0 ;
  for k from 1 to 11 by 2 do
      S :=S+1/k^2 ;
      if S>1.15 then
          break
      fi ; od ;
  S ;
```

$$S := 0$$
$$S := 1$$
$$S := \frac{10}{9}$$
$$S := \frac{259}{225}$$
$$\frac{259}{225}$$

Exercice 4. *Calculer* $n!$ *pour* $n = 3$, $n = 10$, $n = 15$ *à l'aide d'une boucle* `for`*.*

Exercice 5. *On considère la suite* $(u_n)_{n\in\mathbb{N}}$ *définie par :* $\begin{cases} u_0 = 2 \\ u_{n+1} = \dfrac{1}{2}\left(u_n + \dfrac{3}{u_n}\right) \end{cases}$

Calculer u_{10}, u_{20} et u_{100} à l'aide d'une boucle for.
Consigne : On fera du calcul approché.

Exercice 6. *Suite de Syracuse*

On considère la suite $(u_n)_{n\in\mathbb{N}}$ définie par : $u_0 = 31$ et $u_{n+1} = \begin{cases} \dfrac{u_n}{2} & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{si } u_n \text{ est impair} \end{cases}$

Calculer u_{10}, u_{20} et u_{100} à l'aide d'une boucle for.

Exercice 7. *Doubles sommes*

Il est tout à fait possible d'imbriquer deux boucles `for`. Pour cela, il faudra choisir des noms différents pour les variables d'itérations de chacune des boucles `for`.

1. Calculer $\displaystyle\sum_{k=1}^{n}\sum_{p=1}^{n}(k+p)^2$ pour $n = 10$, $n = 20$ et $n = 100$.
2. Calculer $\displaystyle\sum_{k=1}^{n}\sum_{p=1}^{k}(k+p)^2$ pour $n = 10$, $n = 20$ et $n = 100$.