Informatique et Algorithmique – Chapitre 2

Programmation en Scilab

I Introduction à l'algorithmique et la programmation

Un algorithme est une suite finie d'instructions à exécuter dans un ordre déterminé, permettant de résoudre, en un nombre fini d'étapes, un problème particulier.

La structure d'un algorithme est semblable à celle d'une recette de cuisine : on démarre avec une liste d'ingrédients (les entrées pour un algorithme) et on décrit les étapes successives à effectuer dans un certain ordre (les instructions pour un algorithme) à partir des ingrédients et d'outils mis à notre disposition. L'application de ces étapes nous permet d'obtenir des plats (les sorties pour un algorithme) prêts à être servis.

L'outil informatique permet de réaliser rapidement un très grand nombre de calculs. Si chaque étape d'un algorithme donné peut être décomposé en plusieurs sous-étapes suffisamment simples et compréhensibles pour être effectuée par un ordinateur, alors on peut envisager de lui confier l'exécution de cet algorithme. Un programme informatique est donc la traduction d'un algorithme en langage compréhensible par un ordinateur. On dit alors que l'on code ou que l'on implémente l'algorithme.

Il existe de nombreux langages compréhensibles par un ordinateur (*Java, C, Python, Pascal, HTML, LaTeX*, etc). Ils n'ont pas tous la même utilité mais ils respectent généralement les même principes de bases. Dans ce cours, nous utilisons uniquement le langage *Scilab* qui permet l'implémentation d'algorithmes de calcul numérique (cf. chapitre 1).

Exemple : *Un algorithme consistant à écrire un entier naturel en binaire est le suivant :*

1. *On prend en entrée en entier naturel n.*
2. *On stocke la valeur de n dans une variable a.*
3. *On garde en mémoire le reste de la division euclidienne de a par b. On remplace la valeur de a par la valeur du quotient.*
4. *On répète l'étape 2 jusqu'à ce que le quotient de la division soit égal à 0.*
5. *On écrit les restes successifs du dernier au premier.*

Une traduction de cet algorithme en langage Scilab est le programme suivant :

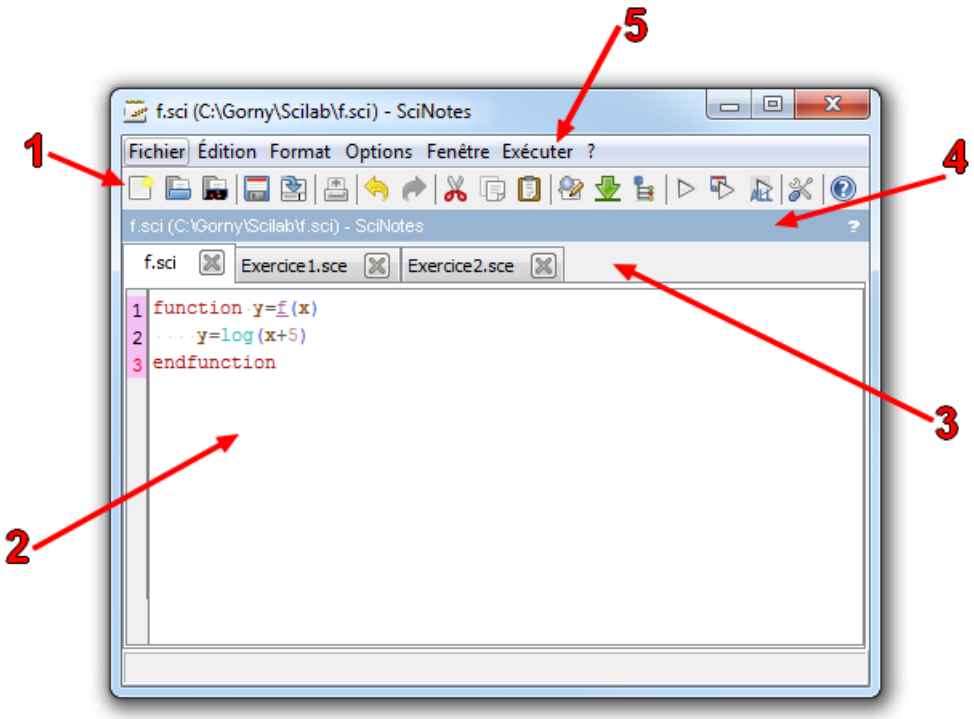
```
//On demande à l'utilisateur d'entrer n et b
n=input('Entrer un entier naturel : ')
b=input('Entrer une base : ')
a=n; q=1; bin='';//On crée une chaine de caractère vide, appelée bin
while  q>0//Tant que q est non nul, on répète les instructions suivantes.
       q=0;
       while  b*(q+1)<a
              q=q+1;
       end//A l'issue de cette boucle, q est le nouveau quotient
       r=a-b*q; a=q;
       bin=string(r)+bin;//On ajoute le reste r devant bin
end
disp('L''entier ' + string(n) + ' s''écrit ' + bin
           + ' en base ' + string(b) + '.')
```

II L'éditeur de texte SciNotes

Lorsque l'on écrit des programmes comportant un grand nombre d'instructions, l'usage de la console Scilab montre vite ses limites. Notamment l'enregistrement n'est pas possible et, à chaque exécution d'une commande, pour la modifier et l'exécuter à nouveau, il faut la réécrire entièrement ou la rechercher dans l'historique. Nous allons donc travailler avec des scripts qui permettent de rassembler en un même fichier une succession d'instructions. Pour créer des scripts, nous allons utiliser l'éditeur de texte **SciNotes**.

SciNotes possède d'autres fonctionnalité très utiles permettent d'améliorer la lisibilité d'un code (organisation par blocs, coloration des mots clés, indentations, complétion automatique pour aider à la rédaction d'un code).

Cliquer sur l'icône ouvre SciNotes dans une nouvelle fenêtre :



1. L'icône permet d'ouvrir un nouveau script. Cliquer sur le menu `Fichier` permet aussi d'ouvrir des fichiers déjà créés et enregistrés.
2. On écrit les instructions du script dans la fenêtre principale. Il est important de penser à enregistrer régulièrement ses scripts, en utilisant le raccourci Ctrl + S .
3. On peut ouvrir plusieurs scripts en même temps qui s'organisent en onglets. Pour accéder à un script déjà ouvert, il suffit de cliquer sur son onglet.
4. En maintenant enfoncé le clic gauche de la souris sur la barre bleue, on peut déplacer SciNotes et l'insérer dans l'environnement de travail Scilab.
5. Un fois un script terminé, on peut l'exécuter dans la console Scilab. Pour cela on peut cliquer sur `Exécuter` et on se retrouve alors face à différents modes d'exécution :
   - l'exécution avec écho (qui affiche dans la console les commandes exécutées et les réponses pour les commandes qui ne sont pas terminées par ;). Un raccourci est Ctrl + L .
   - l'exécution avec écho jusqu'à la position du curseur. Un raccourci est Ctrl + E .
   - l'exécution sans écho (qui affiche seulement les réponses pour les commandes qui ne sont pas terminées par ;). Un raccourci est Ctrl + Maj + E .

   Si le programme est sauvegardé dans le répertoire de travail sous le nom `nomprogramme.sce`, alors on peut aussi exécuter sans écho en tapant sur la console Scilab la commande `exec('nomprogramme.sce', -1)`.

Remarques :

-  Il faut enregistrer un script avant de l'exécuter... sinon on exécute sa précédente version. Dans SciNotes, lorsqu'un script a été modifié mais pas enregistré, une petite astérisque apparait dans le titre de l'onglet (cf. **3**) correspondant au script en question.
- Le symbole // marque le début d'un commentaire : ce qui suit ce symbole ne sera pas exécuté. Un bon code est un code commenté (cela facilite la compréhension du code quand on le reprend quelque temps plus tard...)
- Par défaut, un script est sauvegardé avec l'extension `.sce` (ou `.sci` pour les fonctions). On se retrouvera rapidement avec de nombreux scripts. Pour gagner du temps, il convient donc de les organiser par nom (par exemple `TP3-Exo5.sce`) ou de créer des dossiers.

III Interaction avec l'utilisateur : entrée et sortie de données

Lorsque l'on écrit un programme c'est pour demander à une machine de faire quelque chose. Il est donc important de faciliter la communication avec la machine, c'est-à-dire de savoir utiliser les commandes d'entrées et de sorties.

1) La commande `input`

La commande qui permet à un programme de demander quelque chose à l'utilisateur est `input` (*entrée* en français). Plus précisément, la syntaxe est la suivante :

```
var=input("message")
```

Lors de l'exécution du programme, cette commande va afficher la chaîne de caractères `"message"` sur la console (en général le message demande à l'utilisateur d'écrire une variable : un nombre ou un vecteur ou un booléen, etc.). Ce que l'utilisateur répond au programme (c'est-à-dire la commande qu'il écrit dans la console) est stocké dans la variable `var` et pourra être utilisée par la programme.

Exemple : *(dans la console)*

```
-->n=input("Entrer un nombre entier : ")
Entrer un nombre entier : 6
 n  =
    6.
```

La commande `var=input("message","string")` permet de saisir une chaîne de caractères. Plus précisément, elle affiche comme précédemment la chaîne de caractères `"message"`. Ce que l'utilisateur répond est stocké dans la variable `var` sous la forme d'une chaine de caractère.

Exemple : *(dans la console)*

```
-->nom=input("Comment vous appelez-vous ? ","string")
Comment vous appelez-vous ? Bond, James Bond
 nom  =
 Bond, James Bond
```

2) La commande `disp`

La commande qui permet à un programme d'afficher des variables ou des phrases en sortie est `disp`, abréviation de *display* (*afficher* en français). Plus précisément, la syntaxe est la suivante :

```
disp(var1,var2,...,varn)
```

Ici `var1,var2,...,varn` désignent des variables qui ont été affectées lors des instructions du programme. Il peut s'agit de chaînes de caractères si on veut écrire une phrase.

Exemple : *(dans la console)*

```
-->n=7; s=n*(n+1)/2;
-->disp("La somme des entiers de 1
                à ", n, " est ", s)
   28.
 est
   7.
 La somme des entiers de 1 à
```

```
-->disp(s," est ", n, "La somme
             des entiers de 1 à ")
 La somme des entiers de 1 à
   7.
 est
   28.
```

 Assez curieusement, lors de l'exécution du programme, cette commande affiche les variables dans l'ordre inverse. Le mieux est encore d'utiliser `disp` avec une seule chaîne de caractères qui consiste en une phrase (rappelons que la commande `sting(var)` permet de convertir la variable `var` en chaîne de caractères, cf. le chapitre précédent).

```
-->disp('La somme des entiers de 1 à ' + string(n) + ' est ' + string(s) + '.')
 La somme des entiers de 1 à 7 est 28.
```

Exemple : Écrivons un programme qui demande à l'utilisateur deux réels x et q et un entier n et renvoie la valeur du $n^{\text{ième}}$ terme d'une suite géométrique de raison q et de terme initial x.

```
x=input('Entrer le terme initial de la suite géométrique :')
q=input('Entrer la raison de la suite géométrique :')
n=input('Entrer un rang :')
u=x*q^n;
disp('La valeur du '+string(n)+'-ième terme de la suite géométrique de raison '
        +string(q)+' et de terme initial '+string(x)+' est '+string(u)+'.')
```

Exécutons ce programme (appelé `SuiteGeom.sce`) sans écho dans la console :

```
-->exec('C:\Gorny\Scilab\SuiteGeom.sce', -1)
Entrer le terme initial de la suite géométrique :3
Entrer la raison de la suite géométrique :-1.2
Entrer un rang :19

 La valeur du 19-ième terme de la suite géométrique de raison -1.2
 et de terme initial 3 est -95.844.
```

IV Structures conditionnelles

En programmation, une structure conditionnelle permet de réaliser une séquence d'instructions lorsqu'une certaine condition est vraie (on parle alors de condition `if/then` – *si/alors* en français). Elle peut aussi présenter une alternative et réaliser une autre séquence d'instructions lorsque cette même condition est fausse (on parle alors de condition `if/then/else` – *si/alors/sinon* en français).

La syntaxe dans le cas où il n'y a pas d'alternatives est la suivante :

```
debut_du_programme
if  condition  then
     instructions_if
end
fin_du_programme
```

L'expression `condition` prend pour valeur un booléen (elle soit vraie, soit fausse). Si elle est vraie, alors le bloc d'instructions `instructions_if` s'exécute. Si elle est fausse, alors `instructions_if` est ignoré et c'est directement la suite du programme qui est exécutée.

La syntaxe dans le cas où il y a une alternative est la suivante :

```
debut_du_programme
if  condition  then
      instructions_if
else
      instructions_else
end
fin_du_programme
```

Si `condition` est vraie, alors le bloc d'instructions `instructions_if` s'exécute (mais pas le bloc d'instructions `instructions_else`). Si `condition` est fausse, alors le bloc d'instructions `instructions_else` s'exécute (mais pas `instructions_if`).

Exemple : Écrivons un programme qui demande deux entiers a et b et teste si a est divisible par b.

```
a=input('Entrer un entier : ')
b=input('Entrer un entier non nul : ')
//pour tester si un nombre n est un entier,
//on peut tester s'il est égal à sa partie entière.
n=a/b;
if  n==floor(n)   then
      disp(string(a)+" est divisible par "+string(b)+".")
else
      disp(string(a)+" est divisible par "+string(b)+".")
end
```

Exécutons ce programme (appelé `TestDivisibilite.sce`) sans écho dans la console :

```
-->exec('C:\Gorny\Scilab\TestDivisibilite.sce', -1)
Entrer un entier : 111
Entrer un entier non nul : 11
 111 n'est pas divisible par 11.

-->exec('C:\Gorny\Scilab\TestDivisibilite.sce', -1)
Entrer un entier : 1014
Entrer un entier non nul : 3
 1014 est divisible par 3.
```

Il est également possible d'imbriquer plusieurs structures conditionnelles les unes à l'intérieur des autres. Si on a trois alternatives possibles, une syntaxe possible est la suivante :

```
debut_du_programme
if  condition1  then
      instructions1_if
else
      if  condition2  then
            instructions2
      else
            instructions_else
      end
end
fin_du_programme
```

Les expression `condition1` et `condition2` prennent pour valeur des booléens. Si `condition1` est vraie, alors `instructions1` s'exécute et c'est directement la suite du programme qui est exécutée. Si `condition1` est fausse alors,

- ou bien `condition2` est vraie et alors `instructions2` s'exécute puis la suite du programme.
- ou bien `condition2` est fausse également et alors `instructions_else` s'exécute puis la suite du programme.

Cette solution fonctionne mais il est plus élégant et plus pratique d'utiliser une structure conditionnelle à plusieurs branches avec *elseif*. La syntaxe est la suivante :

```
debut_du_programme
if  condition1  then
      instructions1
elseif  condition2  then
      instructions2
elseif  condition3  then
      instructions3
            ⋮
else
      instructions_else
end
fin_du_programme
```

V Structures répétitives

En programmation, une structure répétitive (ou itérative) permet d'effectuer la répétition d'instructions un certain nombre de fois. Nous allons en distinguer deux : la boucle `for` (*boucle pour* en français) et la boucle `while` (*boucle tant que* en français).

En général, la construction d'une structure répétitive au sein d'un programme se base sur quatre étapes :

1. On identifie les variables d'entrées (qui proviennent du début du programme ou qui sont fournies par l'utilisateur via la commande `input`).
2. On définit des variables qui vont être modifiées lors de la boucle `for` ou `while`. C'est l'étape d'initialisation.
3. On écrit la boucle en tant que telle (`for`$\cdots$`end` ou `while`$\cdots$`end`). Elle utilise les variables d'entrées et, à chaque étape de la boucle, les variables définies à l'étape d'initialisation sont mises à jour.
4. On renvoie les variables de sorties (affichées éventuellement dans une phrase via la commande `disp`) qui sont en général des fonctions des quantités calculées pendant la boucle.

1) Boucle For

Une boucle `for` permet de répéter une séquence d'instructions autant de fois qu'il y a de valeurs dans une certaine liste. La syntaxe est la suivante :

```
debut_du_programme
for  var=liste
     instructions
end
fin_du_programme
```

La variable `var` va prendre successivement toutes les valeurs de la liste `liste` et, pour chacune de ses valeurs, le bloc d'instructions `instructions` va être exécuter.

Remarques :

- Dans la pratique, `liste` est généralement un vecteur du type `m:p:n` avec `m` et `n` des réels et `p` un réel strictement positif. Dans ce cas la variable `var` va prendre successivement toutes les valeurs comprises entre `m` et `n` avec un pas de `p` (si le pas est de 1, alors la commande `m:n` suffit).

- Le bloc d'instructions `instructions` peut dépendre ou pas de la variable `var`.

Exemple : Écrivons un programme qui demande un entier strictement positif n et un entier p et calcule la valeur de la somme $\sum_{i=1}^{n} i^p$.

```
n=input('Entrer un entier strictement positif : ')
p=input('Entrer un entier : ')
s=0;//au départ la somme vaut 0.
for  i=1:n
     s=s+i^p;//à l'étape i, on incrémente la somme de i^p.
end
disp("La somme des puissances "+string(p)+"-ième des entiers de 1 à "
        +string(n)+" est "+string(s)+".")
```

Pour comprendre ce que fait la boucle `for`, on peut s'aider du tableau suivant : si l'utilisateur fournit `n=10` et `p=2`, alors

k	valeur de s en début de boucle	valeur de s en fin de boucle
1	0	$1 = 0 + 1^2$
2	1	$5 = 1 + 2^2 = 0 + 1^2 + 2^2$
3	5	$14 = 5 + 3^2 = 0 + 1^2 + 2^2 + 3^2$
$\vdots$	$\vdots$	$\vdots$
k	$0 + 1^2 \cdots + (k-1)^2$	$0 + 1^2 + \cdots + k^2$
$\vdots$	$\vdots$	$\vdots$
10	285	$385 = 285 + 10^2 = 0 + 1^2 + \cdots + 10^2$

Exécutons ce programme (appelé `SommePuissance.sce`) sans écho dans la console :

```
-->exec('C:\Gorny\Scilab\SommePuissance.sce', -1)
Entrer un entier strictement positif : 15
Entrer un entier : 4
 La somme des puissances 4-ième des entiers de 1 à 15 est 178312.

-->exec('C:\Gorny\Scilab\SommePuissance.sce', -1)
Entrer un entier strictement positif : 127
Entrer un entier : 2
 La somme des puissances 2-ième des entiers de 1 à 127 est 690880.

-->n=127; n*(n+1)*(2*n+1)/6
 ans  =
    690880.
```

2) Boucle While

Par opposition à la boucle `for`, la boucle `while` s'emploie lorsque l'on ne connaît pas à l'avance le nombre d'itérations à effectuer. Elle permet de répéter une séquence d'instructions tant qu'une certaine condition, appelée test d'arrêt, est vraie. Elle s'arrête dès que cette condition est fausse. La syntaxe est la suivante :

```
debut_du_programme
while  condition
       instructions
end
fin_du_programme
```

L'expression `condition` prend pour valeur un booléen (elle soit vraie, soit fausse). Tant qu'elle est vraie le bloc d'instruction `instructions` va être exécuter.

Remarques :

- Les instructions peuvent ne pas être exécutées du tout, si dès le départ `condition` a la valeur `T`.
- Il se peut aussi que la boucle ne s'arrête jamais si `condition` ne prend jamais la valeur `F` : on parle alors de boucle infinie. Dans ce cas il faudra utiliser Ctrl + C pour arrêter la programme. Pour éviter que cela arrive, il est impératif que `condition` dépendent de plusieurs variables qui sont modifiées lors des exécutions successives de `instructions`.

Exemple : Pour tout $n \in \mathbb{N}^*$, posons $H_n = \sum_{k=1}^{n} \frac{1}{k}$. La suite $(H_n)_{n\in\mathbb{N}^*}$ est croissante et tend vers $+\infty$.

Écrivons un programme qui demande à l'utilisateur un réel strictement positif A et détermine le plus petit rang $n \in \mathbb{N}^*$ pour lequel $u_n \geqslant A$.

```
n=input('Entrer un réel A strictement positif : ')
n=1; H=1;//le rang initial est n=1 la somme H_n vaut 1.
while  H<A
       n=n+1;//Tant que H<A, on passe au rang suivant.
       H=H+1/n;//et on met à jour la valeur de H.
end
disp("Les termes de la suites sont supérieurs ou égaux à "
          + string(A) + " à partir du rang " + string(n) + ".")
```

Pour comprendre ce que fait la boucle `while`, on peut s'aider du tableau suivant : si l'utilisateur fournit `A`=3, alors

valeurs en début de boucle		H<A	valeurs en fin de boucle	
n	H		n	H
1	1	vrai	2	1.5
2	1.5	vrai	3	1.8333333
3	1.8333333	vrai	4	2.0833333
4	2.0833333	vrai	5	2.2833333
5	2.2833333	vrai	6	2.45
6	2.45	vrai	7	2.5928571
7	2.5928571	vrai	8	2.7178571
8	2.7178571	vrai	9	2.8289683
9	2.8289683	vrai	10	2.9289683
10	2.8289683	vrai	11	3.0198773
11	3.0198773	faux	11	3.0198773

Exécutons ce programme (appelé `SommeHarmonique.sce`) sans écho dans la console :

```
-->exec('C:\Gorny\Scilab\SommeHarmonique.sce', -1)
Entrer un réel A strictement positif : 3
 Les termes de la suites sont supérieurs ou égaux à 3 à partir du rang 11.

-->exec('C:\Gorny\Scilab\SommeHarmonique.sce', -1)
Entrer un réel A strictement positif : 10
  Les termes de la suites sont supérieurs ou égaux à 10 à partir du rang 12367.

-->exec('C:\Gorny\Scilab\SommeHarmonique.sce', -1)
Entrer un réel A strictement positif : 15
  Les termes de la suites sont supérieurs ou égaux à 15 à partir du rang 1835421.
```

Cette boucle `while` va forcément s'arrêter puisque la suite tend vers $+\infty$. Par contre l'exécution du programme peut prendre beaucoup de temps même pour des valeurs faibles de A (essayez par exemple $A = 20$).