1. Introduction à l'environnement SCILAB

1.1. Introduction

SCILAB est un langage de programmation doté d'une riche collection d'algorithmes couvrants plusieurs aspects du calcul scientifique.

- **Du point de vue logiciel**, SCILAB est un langage interprété. Cela permet des processus de développements plus rapides.
- **Du point de vue licence**, SCILAB est gratuit et « Open Source ». Il existe des versions pour les systèmes d'exploitations : Linux, MacOsX et Windows.
- **Du point de vue scientifique**, SCILAB couvre, par exemple, ces domaines :
  - algèbre linéaire, matrices vides
  - polynômes et fonctions rationnelles
  - interpolation, approximation
  - optimisation linéaire, quadratique et non linéaire
  - résolution d'équations différentielles ordinaires et résolution d'équations différentielles algébriques
  - contrôle robuste et classique, optimisation inégalité matricielle linéaire
  - optimisation différentielle et non différentielle
  - traitement du signal
  - statistiques

1.2. L'environnement SCILAB

Lorsque **SCILAB** est lancé, un groupement de 4 fenêtres apparaît à l'écran. L'environnement de travail se compose du navigateur de fichiers, de la console, du navigateur de variables et de l'historique des commandes.

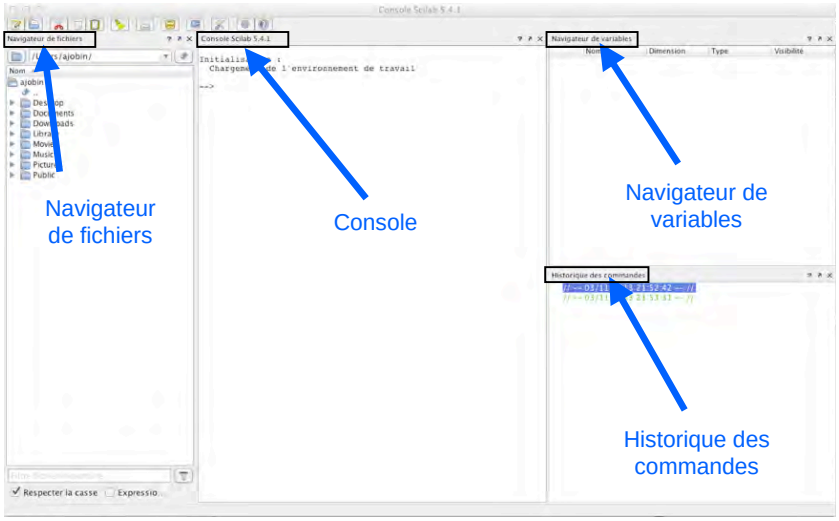


Figure 1 : Environnement de travail

● Le navigateur de fichiers permet de se déplacer dans l'arborescence des fichiers de l'ordinateur.

● Le navigateur de variables permet de visionner les valeur des variables utilisées.

● L'historique des commandes permet de visualiser l'ensemble des commandes qui ont été précédemment tapées.

● La console permet l'interprétation directe des commandes que l'on tape. On utilise ainsi **SCILAB** comme une super calculatrice.

1.2.1. Première interaction avec SCILAB

Le moyen le plus simple d'utiliser SCILAB est d'écrire directement dans la fenêtre de console juste après le curseur (prompt) **-->**

Pour calculer une expression mathématique il suffit de l'écrire comme ceci :

```
--> 5+6                                  Puis on clique sur la touche Entrer pour voir le résultat
      ans =
            11.
```

Si nous voulons qu'une expression soit calculée mais sans afficher le résultat, on ajoute un point-virgule '**;**' à la fin de l'expression comme suit :

```
--> 5+6 ;
```

Pour créer une variable on utilise la structure simple : '**variable = définition**' sans se préoccuper du type de la variable.

Par exemple :

```
--> a = 10 ;
--> u = cos(a) ;
--> v = sin(a) ;
--> u^2+v^2
      ans =
            1.
--> ans+10
      ans =
            11.
```

Il est possible d'écrire plusieurs expressions dans la même ligne en les faisant séparées par des virgules ou des points virgules. Par exemple :

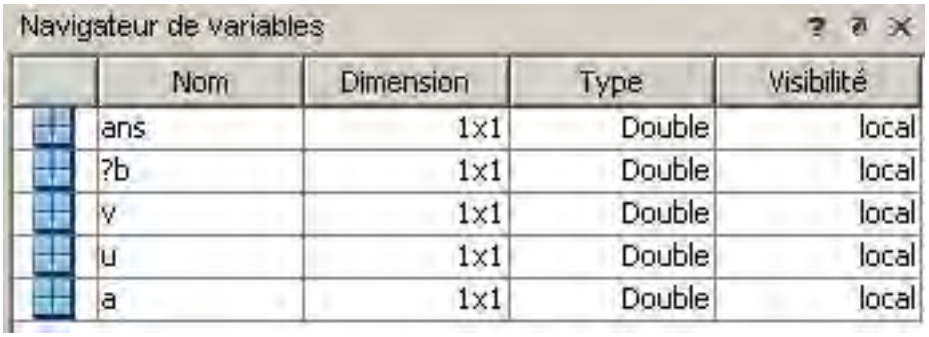
```
--> 5+6, 2*5-1, 12-4
      ans =
            11.
      ans =
            9.
      ans =
            8.
```

```
--> 5+6; 2*5-1, 12-4;
      ans =
            9.
```

Les noms des variables peuvent être aussi long que l'on veut, mais ne comportent que des caractères alpha-numériques et les caractères « %, _, #, !, $, ? ». Les variables commençant par % sont des variables prédéfinies en SCILAB. Nous devons aussi faire attention aux majuscules car le SCILAB est sensible à la casse (**A** et **a** sont deux identifiants différents). Les opérations de base dans une expression sont résumées dans le tableau suivant :

L'opération	La signification
+	L'addition
-	La soustraction
*	La multiplication
/	La division
\	La division gauche (ou la division inverse)
^ ou **	La puissance
'	Le transposé
(et)	Les parenthèses spécifient l'ordre d'évaluation

Les informations sur les variables sont visibles directement dans la fenêtre navigateur de variables.



1.2.2. Les nombres en SCILAB

SCILAB utilise une notation décimale conventionnelle, avec un point décimal obligatoire '.' et le signe '+' ou '–' pour les nombres signés. La notation scientifique utilise la lettre 'e' pour spécifier le facteur d'échelle en puissance de 10. Les nombres complexes utilise les caractères '%i' pour designer la partie imaginaire. Le tableau suivant donne un résumé :

Le type	Exemples
Entier	5 -83
Réel en notation décimale	0.0205 3.1415926
Réel en notation scientifique	1.60210e-20 6.02252e23 (1.60210×10^{-20} et 6.02252×10^{23})
Complexe	5+3*%i -3.14159*%i

SCILAB utilise toujours les nombres réels (double précision) pour faire les calculs, ce qui permet d'obtenir une précision de calcule allant jusqu'aux 16 chiffres significatifs. Le résultat d'une opération de calcule est par défaut affichée avec sept chiffres après la virgule.

La fonction **format** peut être utilisé afin de forcer le calcule de présenter plus de décimaux significatifs en spécifiant le nombre de décimaux désirés.

Exemple :

```
--> sqrt(2)
      ans =
           1.4142136
--> format(25)
--> sqrt(2)
      ans =
           1.4142135623730951454746
```

1.2.3. Les principales constantes, fonctions et commandes

SCILAB définit les constantes suivantes :

La constante	Sa valeur
%pi	π=3.1415...
exp(1)	e=2.7183...
%i	$=\sqrt{-1}$
%inf	∞
%nan	Not a Number (Pas un numéro)
%eps	$\varepsilon \approx 2 \times 10^{-16}$.

Parmi les fonctions fréquemment utilisées, on peut noter les suivantes :

La fonction	Sa signification
sin(x)	le sinus de x (en radian)
cos(x)	le cosinus de x (en radian)
tan(x)	le tangent de x (en radian)
asin(x)	l'arc sinus de x (en radian)
acos(x)	l'arc cosinus de x (en radian)
atan(x)	l'arc tangent de x (en radian)
sqrt(x)	la racine carrée de x ➔ $\sqrt{x}$
abs(x)	la valeur absolue de x ➔ |x|
exp(x)	= e^x
log(x)	logarithme naturel de x ➔ ln(x)=$\log_e$(x)
log10(x)	logarithme à base 10 de x ➔ $\log_{10}$(x)
imag(x)	la partie imaginaire du nombre complexe x
real(x)	la partie réelle du nombre complexe x
round(x)	arrondi un nombre vers l'entier le plus proche
floor(x)	arrondi un nombre vers l'entier le plus petit ➔ min{n|n≤x, n entier}
ceil(x)	arrondi un nombre vers l'entier le plus grand ➔ max{n|n≥x, n entier}

SCILAB offre beaucoup de commandes pour l'interaction avec l'utilisateur. Nous nous contentons pour l'instant d'un petit ensemble, et nous exposons les autres au fur et à mesure de l'avancement du cours.

La commande	Sa signification
who	Affiche le nom des variables utilisées
whos	Affiche des informations sur les variables utilisées
clear x y	Supprime les variables x et y
clear, clear all	Supprime toutes les variables
clc	Efface l'écran des commandes
exit, quit	Fermer l'environnement SCILAB
format	Définit le format de sortie pour les valeurs numériques

1.2.4. La priorité des opérations dans une expression

L'évaluation d'une expression s'exécute de gauche à droite en considérant la priorité des opérations indiquée dans le tableau suivant :

Les opérations	La priorité (1=max, 4=min)
Les parenthèses (et)	1
La puissance et le transposé ^ et '	2
La multiplication et la division * et /	3
L'addition et la soustraction + et -	4

Par exemple : **5+2*3 = 11.** et **2*3^2 = 18.**

Exercice récapitulatif :

Créer une variable x et donnez-la la valeur 2, puis écrivez les expressions suivantes :

- $3x^3-2x^2+4x$
- $\frac{e^{1+x}}{1-\sqrt{2x}}$
- $|\sin^{-1}(2x)|$
- $\frac{\ln(x)}{2x^3}$ -1

La solution :

```
--> x=2 ;
--> 3*x^3-2*x^2+4*x ;
--> exp(1+x)/(1-sqrt(2*x)) ;
--> abs(asin(2*x)) ;
--> log(x)/(2*x^3)-1 ;
```

2. Les vecteurs et les matrices

SCILAB était conçu à l'origine pour permettre aux mathématiciens, scientifiques et ingénieurs d'utiliser facilement les mécanismes de l'algèbre linéaire. Par conséquent, l'utilisation des vecteurs et des matrices est très intuitif et commode en SCILAB.

2.1. Les vecteurs

Un vecteur est une liste ordonnée d'éléments. Si les éléments sont arrangés horizontalement on dit que le vecteur est un vecteur ligne, par contre si les éléments sont arrangés verticalement on dit que c'est un vecteur colonne.

Pour créer un **vecteur ligne** il suffit d'écrire la liste de ses composants entre crochets **[** et **]** et de les séparés par des espaces ou des virgules comme suit :

```
--> V = [ 5 , 2 , 13 , -6 ]              \\ Création d'un vecteur ligne V
      V =
               5.    2.   13.   -6.

--> U = [ 4 -2 1 ]                       \\ Création d'un vecteur ligne U
      U =
               4.   -2.    1.
```

Pour créer un **vecteur colonne** il est possible d'utiliser une des méthodes suivantes :

1. écrire les composants du vecteur entre crochets **[** et **]** et de les séparés par des points-virgules (**;**) comme suit :

```
--> U = [ 4 ; -2 ; 1 ]                  \\ Création d'un vecteur colonne U
      U =
              4.
             -2.
              1.
```

2. calculer le transposé d'un vecteur ligne :

```
--> U = [ 4 -2 1 ]'                      \\ Création d'un vecteur colonne U
      U =
              4.
             -2.
              1.
```

Si les composants d'un vecteur **X** sont ordonnés avec des valeurs consécutives, nous pouvons le noter avec la notation suivante :

```
X = premier_élément : dernier_élément
```

(Les crochets sont facultatifs dans ce cas)

Par exemple :

```
--> X = 1:8
      X =
           1.    2.    3.    4.    5.    6.    7.    8.
```

```
--> X = [1:8]
      X =
            1.    2.    3.    4.    5.    6.    7.    8.
```

Si les composants d'un vecteur **X** sont ordonnés avec des valeurs consécutives mais avec un pas (d'incrémentation/décrémentation) différente de 1, nous pouvons spécifier le pas avec la notation :

X = premier_élément : le_pas : dernier_élément (Les crochets sont facultatifs)

Par exemple :

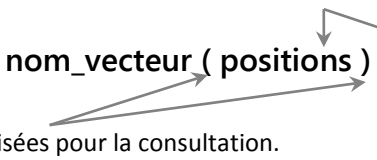
```
--> X = [0:2:10]                \\ le vecteur X contient les nombres pairs < 12
      X =
            0.    2.    4.    6.    8.   10.
--> X = [-4:2:6]
      X =
            -4.   -2.    0.    2     4.    6.
--> X = 0:0.2:1
      X =
            0.     0.2     0.4     0.6    0.8     1.
```

On peut écrire des expressions plus complexes comme :

```
--> V = [ 1:2:5 , -2:2:1 ]
      V =
            1.    3.    5.   -2.    0.
--> A = [1 2 3]
      A =
            1.    2.    3.
--> B = [A, 4, 5, 6]
      B =
            1.    2.    3.    4.    5.    6.
```

2.1.1. Référencement et accès aux éléments d'un vecteur

L'accès aux éléments d'un vecteur se fait en utilisant la syntaxe générale suivante :



Les parenthèses **(** et **)** sont utilisées pour la consultation.
Les crochets **[** et **]** sont utilisés uniquement pendant la création.

positions : peut être un simple numéro, ou une liste de numéro (un vecteur de positions)

Exemples :

```
--> V = [5, -1, 13, -6, 7]      \\ création du vecteur V qui contient 5 éléments
      V =
             5.   -1.   13.   -6.    7.
--> V(3)                         \\ la 3ème position
      ans =
            13.
```

```
--> V(2:4)                          \\ de la deuxième position jusqu'au quatrième
      ans =
            -1.   13.   -6.

--> V(4:-2:1)                       \\ de la 4ème pos jusqu'à la 1ère avec le pas = -2
      ans =
            -6.   -1.

--> V(3:$)                          \\ de la 3ème position jusqu'à la dernière
      ans =
            13.   -6.    7.

--> V([1,3,4])                      \\ la 1ére, la 3ème et la 4ème position uniquement
      ans =
            5.   13.   -6.

--> V(1) = 8                        \\ donner la valeur 8 au premier élément
      V =
            8.   -1.   13.   -6.    7.

--> V(6) = -3                       \\ ajouter un sixième élément avec la valeur -3
      V =
            8.   -1.   13.   -6.    7.   -3.

--> V(9) = 5                        \\ ajouter un neuvième élément avec la valeur 5
      V =
            8.   -1.   13.   -6.    7.   -3.    0.    0.    5.

--> V(2) = []                       \\ Supprimer le deuxième élément
      V =
            8.   13.  -6.    7.   -3.    0.    0.    5.

--> V(3:5) = []                     \\ Supprimer du 3ème jusqu'au 5ème élément
      V =
            8.   13.   0.    0.    5.
```

2.1.2. Les opérations élément par élément pour les vecteurs

Avec deux vecteurs $\vec{u}$ et $\vec{v}$, il est possible de réaliser des calcules élément par élément en utilisant les opérations suivantes :

L'opération	**Signification**	**Exemple avec :** **--> u = [-2, 6, 1] ;** **--> v = [3, -1, 4] ;**
+	Addition des vecteurs	**--> u+2** ans = 0. 8. 3. **--> u+v** ans = 1. 5. 5.

-	Soustraction des vecteurs	**--> u-2** ans = -4. 4. -1. **--> u-v** ans = -5. 7. -3.
.*	Multiplication élément par élément	**--> u*2** ans = -4. 12. 2. **--> u.*2** ans = -4. 12. 2. **--> u.*v** ans = -6. -6. 4.
./	Division élément par élément	**--> u/2** ans = -1. 3. 0.5 **--> u./2** ans = -1. 3. 0.5 **--> u./v** ans = -0.6666667 -6. 0.25
.^	Puissance élément par élément	**--> u.^2** ans = 4. 36. 1. **--> u.^v** ans = -8. 0.1666667 1.

L'écriture d'une expression tel que : **u^2** génère une erreur car cette expression réfère a une multiplication de matrices (**u*u** doit être réécrite **u*u'** ou **u'*u** pour être valide).

2.1.3. La fonction linspace

La création d'un vecteur dont les composants sont ordonnés par intervalle régulier et avec un nombre d'éléments bien déterminé peut se réaliser avec la fonction :

linspace (début, fin, nombre d'éléments)

Le pas d'incrémentation est calculé automatiquement par SCILAB selon la formule :

$$le\ pas = \frac{fin - debut}{nombre\ \ 'éléments - 1}$$

Par exemple :

```
--> X = linspace(1,10,4)          \\ un vecteur de quatre élément de 1 à 10
      X =
            1.    4.    7.    10.

--> Y = linspace(13,40,4)         \\ un vecteur de quatre élément de 13 à 40
      Y =
            13.   22.   31.   40.
```

La taille d'un vecteur (le nombre de ses composants) peut être obtenue avec la fonction **length** comme suit :

```
--> length(X)                     \\ la taille du vecteur X
      ans =
            4.
```

2.2. Les matrices

Une matrice est un tableau rectangulaire d'éléments (bidimensionnels). Les vecteurs sont des matrices avec une seule ligne ou une seule colonne (monodimensionnels). Pour insérer une matrice, il faut respecter les règles suivantes :

- Les éléments doivent être mises entre des crochets **[** et **]**
- Les espaces ou les virgules sont utilisés pour séparer les éléments dans la même ligne
- Un point-virgule est utilisé pour séparer les lignes

Pour illustrer cela, considérant la matrice suivante :

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

Cette matrice pout être écrite en SCILAB avec une des syntaxes suivantes :

```
--> A = [1,2,3,4 ; 5,6,7,8 ; 9,10,11,12] ;
```

```
--> A = [1 2 3 4 ; 5 6 7 8 ; 9 10 11 12] ;
```

```
--> A = [[1;5;9] , [2;6;10] , [3;7;11] , [4;8;12]] ;
```

Le nombre d'éléments dans chaque ligne (nombre de colonnes) doit être identique dans toutes les lignes de la matrice, sinon une erreur sera signalée par SCILAB.
Par exemple :

```
--> X = [1 2 ; 4 5 6]
      Dimensions rangée/colonne incohérentes.
```

Une matrice peut être générée par des vecteurs comme le montre les exemples suivants :

```
--> x = 1:4                          \\ création d'un vecteur x
      x =
            1.    2.    3.    4.
--> y = 5:5:20                       \\ création d'un vecteur y
      y =
            5.   10.   15.   20.
--> z = 4:4:16                       \\ création d'un vecteur z
      z =
            4.    8.   12.   16.

--> A = [x ; y ; z]                  \\ A est formée par les vecteurs lignes x, y et z
      A =
            1.    2.     3.     4.
            5.    10.    15.    20.
            4.    8.     12.    16.

--> B = [x' y' z']                   \\ B est formée par les vecteurs colonnes x, y et z
```

```
      B  =
            1.    5.     4.
            2.    10.    8.
            3.    15.    12.
            4.    20.    16.

--> C = [x ; x]                \\ C est formée par le même vecteur ligne x 2 fois
      C =
            1.    2.    3.     4.
            1.    2.    3.     4.
```

2.2.1. Référencement et accès aux éléments d'une matrice

L'accès aux éléments d'une matrice se fait en utilisant la syntaxe générale suivante :

nom_matrice (positions_lignes , positions_colonnes)

positions : peut être un simple numéro, ou une liste de numéro (un vecteur de positions)

Les parenthèses **(** et **)** sont utilisées pour la consultation).
Les crochets **[** et **]** sont utilisés uniquement pendant la création.

Il est utile de noter les possibilités suivantes :

- L'accès à un élément de la ligne **i** et la colonne **j** se fait par : **A(i,j)**
- L'accès à toute la ligne numéro **i** se fait par : **A(i,:)**
- L'accès à toute la colonne numéro **j** se fait par : **A(:,j)**

Exemples :

```
--> A = [1,2,3,4 ; 5,6,7,8 ; 9,10,11,12]     \\ création de la matrice A
      A =
            1.    2.    3.    4.
            5.    6.    7.    8.
            9.   10.   11.   12.

--> A(2,3)                      \\ l'élément sur la 2ème ligne à la 3ème colonne
      ans =
            7.

--> A(1,:)                      \\ tous les éléments de la 1ère ligne
      ans =
            1.    2.    3.    4.

--> A(:,2)                      \\ tous les éléments de la 2ème colonne
      ans =
            2.
            6.
            10.

--> A(2:3,:)                    \\ tous les éléments de la 2ème et la 3ème ligne
      ans =
            5.    6.    7.    8.
            9.   10.   11.   12.
```

```
--> A(1:2,3:4)                    \\ La sous matrice supérieure droite de taille 2x2
      ans =
            3.    4.
            7.    8.

--> A([1,3],[2,4])                \\ la sous matrice : lignes(1,3) et colonnes (2,4)
      ans =
            2.    4.
           10.   12.

--> A(:,3) = []                   \\ Supprimer la troisième colonne
      A =
            1.    2.   4.
            5.    6.   8.
            9.   10.  12.

--> A(2,:) = []                   \\ Supprimer la deuxième ligne
      A =
            1.    2.   4.
            9.   10.  12.

--> A = [A , [0;0]]               \\ Ajouter une nouvelle colonne avec A(:,4)=[0;0]
      A =
            1.    2.    4.    0.
            9.   10.   12.    0.

--> A = [A ; [1,1,1,1]]           \\ Ajouter une nouvelle ligne avec A(3,:)=[1,1,1,1]
      A =
            1.    2.    4.    0.
            9.   10.   12.    0.
            1.    1.    1.    1.
```

Les dimensions d'une matrice peuvent être acquises en utilisant la fonction **size**. Cependant, avec une matrice A de dimension **m × n** le résultat de cette fonction est un vecteur de deux composants, une pour **m** et l'autre pour **n**.

```
--> d = size(A)
      d =
            3.    4.
```

Ici, la variable **d** contient les dimensions de la matrice A sous forme d'un vecteur. Pour obtenir les dimensions séparément on peut utiliser la syntaxe :

```
--> d1 = size (A, 1)              \\ d1 contient le nombre de ligne (m)
      d1 =
            3.

--> d2 = size (A, 2)              \\ d2 contient le nombre de colonne (n)
      d2 =
            4.
```

2.2.2. Génération automatique des matrices

En SCILAB, il existe des fonctions qui permettent de générer automatiquement des matrices particulières. Dans le tableau suivant nous présentons-les plus utilisées :

La fonction	Signification
zeros(m,n)	Génère une matrice m × n avec tous les éléments = 0
ones(m,n)	Génère une matrice m × n avec tous les éléments = 1
eye(m,n)	Génère une matrice identité de dimension m × n
rand(m,n)	Génère une matrice de dimension m × n de valeurs aléatoires

2.2.3. Les opérations de base sur les matrices

L'opération	Signification
+	L'addition
-	La soustraction
.*	La multiplication élément par élément
./	La division élément par élément
.	La division inverse élément par élément
.^	La puissance élément par élément
*****	La multiplication matricielle
/	La division matricielle (A/B) = ($A*B^{-1}$)

Les opérations élément par éléments sur les matrices sont les mêmes que ceux pour les vecteurs (la seule condition nécessaire pour faire une opération élément par élément est que les deux matrices aient les mêmes dimensions). Par contre la multiplication ou la division des matrices requiert quelques contraintes (consulter un cours sur l'algèbre matricielle pour plus de détail).

Exemple :

```
--> A=ones(2,3)
 A =
    1.    1.    1.
    1.    1.    1.
--> B=zeros(3,2)
 B =
    0.    0.
    0.    0.
    0.    0.
--> B=B+3
 B =
    3.    3.
    3.    3.
    3.    3.
--> A*B
 ans =
    9.    9.
    9.    9.
```

```
--> B=[B , [3 3 3]']          \\ ou bien B(:,3)=[3 3 3]'
      B =
            3.    3.    3.
            3.    3.    3.
            3.    3.    3.
--> B=B(1:2,:)                \\ ou bien B(3,:)=[]
      B =
            3.    3.    3.
            3.    3.    3.
--> A=A*2
      A =
            2.    2.    2.
            2.    2.    2.

--> A.*B
         ans =
           6.    6.    6.
           6.     6.
6. --> A*eye(2,3)
       ans =
           2.    2.    2.
           2.    2.    2.
```

2.2.4. Fonctions utiles pour le traitement des matrices

Voici quelques fonctions parmi les plus utilisées concernant les matrices :

La fonction	L'utilité	Exemple d'utilisation

det	Calcule de déterminant d'une matrice	`--> A = [1,2;3,4] ;`
		`--> det(A)`
		`ans =`
		`-2.`

inv	Calcule l'inverse d'une matrice	`--> inv(A)`
		`ans =`
		`-2.  1.`
		`1.5  -0.5`

rank	Calcule le rang d'une matrice	`--> rank(A)`
		`ans =`
		`2.`

trace	Calcule la trace d'une matrice	`--> trace(A)`
		`ans =`
		`5.`

norm	Calcule la norme d'un vecteur	`ans =` `3.`

diag	Renvoie le diagonal d'une matrice	`ans =`
		`1.`
		`4.`

diag(V)	Crée une matrice ayant le vecteur V dans le diagonal et 0 ailleurs.	`--> V = [-5,1,3]`
		`--> diag(V)`
		`ans =`
		`-5.  0.  0.`
		`0.  1.  0.`
		`0.  0.  3.`

tril	Renvoie la partie triangulaire inferieure	--> B=[1,2,3;4,5,6;7,8,9] B = 1. 2. 3. 4. 5. 6. 7. 8. 9. --> tril(B) ans = 1. 0. 0. 4. 5. 0. 7. 8. 9. --> tril(B,-1) ans = 0. 0. 0. 4. 0. 0. 7. 8. 0. --> tril(B,-2) ans = 0. 0. 0. 0. 0. 0. 7. 0. 0.
triu	Renvoie la partie triangulaire supérieure	--> triu(B) ans = 1. 2. 3. 0. 5. 6. 0. 0. 9. --> triu(B,-1) ans = 1. 2. 3. 4. 5. 6. 0. 8. 9. --> triu(B,1) ans = 0. 2. 3. 0. 0. 6. 0. 0. 0.

3. Introduction à la programmation avec SCILAB

Nous avons vu jusqu'à présent comment utiliser SCILAB pour effectuer des commandes ou pour évaluer des expressions en les écrivant dans la ligne de commande (Après le prompt **-->**), par conséquent les commandes utilisées s'écrivent généralement sous forme d'une seule instruction (éventuellement sur une seule ligne).

Cependant, il existe des problèmes dont la description de leurs solutions nécessite plusieurs instructions, ce qui réclame l'utilisation de plusieurs lignes. Comme par exemple la recherche des racines d'une équation de second degré (avec prise en compte de tous les cas possibles).

Une collection d'instructions bien structurées visant à résoudre un problème donnée s'appelle un programme. Dans cette partie du cours, on va présenter les mécanismes d'écriture et d'exécution des programmes en SCILAB.

3.1. Généralités

3.1.1. Les commentaires

Les commentaires sont des phrases explicatives ignorées par SCILAB et destinées pour l'utilisateur afin de l'aider à comprendre la partie du code commentée.

En SCILAB un commentaire commence par le symbole **//** et occupe le reste de la ligne.

Par exemple :

```
--> A=B+C ; // Donner à A la valeur de B+C
```

3.1.2. Écriture des expressions longues

Si l'écriture d'une expression longue ne peut pas être enclavée dans une seule ligne, il est possible de la diviser en plusieurs lignes en mettant à la fin de chaque ligne au moins trois points.

Exemple :

```
--> (sin(pi/3)^2/cos(pi/3)^2)-(1-2*(5+sqrt(x)^5/(-2*x^3-x^2)^1+3*x)) ;
```

Cette expression peut être réécrite de la façon suivante :

```
--> (sin(pi/3)^2/cos(pi/3)^2)- ...  ↵
--> (1-2*(5+sqrt(x)^5 .....         ↵
--> /(-2*x^3-x^2)^1+3*x)) ;         ↵
```

3.1.3. Lecture des données dans un programme (Les entrées)

Pour lire une valeur donnée par l'utilisateur, il est possible d'utiliser la commande **input**, qui a la syntaxe suivante :

```
variable = input ('une phrase indicative')
```

La valeur déposée par l'utilisateur sera mise dans cette variable

Une phrase aidant l'utilisateur à savoir quoi entrer

Quand SCILAB exécute une telle instruction, La phrase indicative sera affichée à l'utilisateur en attendant que ce dernier entre une valeur.

Par exemple :

```
--> A = input ('Entrez un nombre entier : ')          ↵
   Entrez un nombre entier : 5                        ↵
   A =
       5.
--> A = input ('Entrez un nombre entier : ');         ↵
   Entrez un nombre entier : 5                        ↵
--> B = input ('Entrez un vecteur ligne : ')          ↵
   Entrez un vecteur ligne : [1:2:8,3:-1:0]           ↵
   B =
       1.    3.    5.    7.    3.    2.    1.    0.
```

3.1.4. Ecriture des données dans un programme (Les sorties)

On a déjà vu que SCILAB peut afficher la valeur d'une variable en tapant seulement le nom de cette derniére.

Par exemple :

```
--> A = 5 ;
--> A                  // Demander à SCILAB d'afficher la valeur de A
   A =
      5.
```

Avec cette méthode, SCILAB écrit le nom de la variable (A) puis le signe (=) suivie de la valeur désirée. Cependant, il existe des cas où on désire afficher uniquement la valeur de la variable (sans le nom et sans le signe =).

Pour cela, on peut utiliser la fonction **disp**, et qui a la syntaxe suivante : **disp (objet)**

La valeur de l'objet peut être un nombre, un vecteur, une matrice, une chaine de caractères ou une expression.

On signale qu'avec un vecteur ou une matrice vide, **disp** n'affiche rien.

Exemple :

```
--> disp(A)                  // Afficher la valeur de A sans 'A = '
      5.
--> disp(A);                 // Le point virgule n'a pas d'effet
      5.
--> B                        // Afficher le vecteur B par la méthode classique
   B =
      1.    3.    5.    7.    3.    2.    1.    0.
--> disp(B)                  // Afficher le vecteur B sans 'B = '
      1.    3.    5.    7.    3.    2.    1.    0.
--> C = 3:1:0                // Création d'un vecteur C vide
   C =
      []
--> disp(C)                  // disp n'affiche rien si le vecteur est vide
      []
```

3.2. Les expressions logiques

3.2.1. Les opérations logiques

L'opération de comparaison	Sa signification
==	l'égalité
~=	l'inégalité
>	supérieur à
<	inferieur à
>=	supérieur ou égale à
<=	inferieur ou égale à
L'opération logique	**Sa signification**
&	le **et** logique
|	le **ou** logique
~	la **négation** logique

En SCILAB une variable logique peut prendre les valeurs **1** (vrai) ou **0** (faux) avec une petite règle qui admette que :

1) Toute valeur égale à **0** sera considérée comme fausse **(= 0 ⇒ Faux)**

2) Toute valeur différente de **0** sera considérée comme vrai **(≠ 0 ⇒ Vrai)**.

Le tableau suivant résume le fonctionnement des opérations logiques :

a	b	a & b	a | b	~a
1 (vrai)	1 (vrai)	T	T	F
1 (vrai)	0 (faux)	F	T	F
0 (faux)	1 (vrai)	F	T	T
0 (faux)	0 (faux)	F	F	T

Par exemple :

```
--> x=10;
--> y=20;
--> x < y                          // affiche T (vrai)
      ans =
            T
--> x <= 10                        // affiche T (vrai)
      ans =
            T
--> x == y                         // affiche F (faux)
      ans =
            F
--> (0 < x) & (y < 30)             // affiche T (vrai)
      ans =
            T
--> (x > 10) | (y > 100)           // affiche F (faux)
      ans =
            F
```

```
--> ~(x > 10)                    // affiche T (vrai)
      ans =
            T
--> 10 & 1                       // 10 est considéré comme vrai donc 1 & 1 = T
      ans =
            T
--> 10 & 0                       // 1 & 0 = F
      ans =
            F
```

3.2.2. Comparaison des matrices

La comparaison des vecteurs et des matrices diffère quelque peu des scalaires, d'où l'utilité des deux fonctions '**isequal**' et '**isempty**' (qui permettent de donner une réponse concise pour la comparaison).

La fonction	Description
isequal	teste si deux (ou plusieurs) matrices sont égales (ayant les mêmes éléments partout). Elle renvoie **T** si c'est le cas, et **F** sinon.
isempty	teste si une matrice est vide (ne contient aucun élément). Elle renvoie **T** si c'est le cas, et **F** sinon.

Pour mieux percevoir l'impact de ces fonctions suivons l'exemple suivant :

```
--> A = [5,2;-1,3]          // Créer la matrice A
      A =
            5.    2.
           -1.    3.
--> B = [5,1;0,3]           // Créer la matrice B
      B =
            5.    1.
            0.    3.
--> A==B                    // Tester si A=B ? (T ou F selon la position)
      ans =
            T     F
            F     T
--> isequal(A,B)           // Tester si effectivement A et B sont égales (les mêmes)
      ans =
            F
--> C=[] ;                  // Créer la matrice vide C
--> isempty(C)              // Tester si C est vide (affiche vrai = T)
      ans =
            T
--> isempty(A)              // Tester si A est vide (affiche faux = F)
      ans =
            F
```

3.3. Structures de contrôle de flux

Les structures de contrôle de flux sont des instructions permettant de définir et de manipuler l'ordre d'exécution des tâches dans un programme. Elles offrent la possibilité de réaliser des traitements différents selon l'état des données du programme, ou de réaliser des boucles répétitives pour un processus donnée.

SCILAB compte huit structures de contrôle de flux à savoir :

- **if**
- **select**
- **for**
- **while**
- **continue**
- **break**
- **try - catch**
- **return**

Nous exposons les quatre premières : (**if, select, for** et **while**)

3.3.1. L'instruction if

L'instruction **if** est la plus simple et la plus utilisée des structures de contrôle de flux. Elle permette d'orienter l'exécution du programme en fonction de la valeur logique d'une condition. Sa syntaxe générale est la suivante :

```
if (condition)
    instruction_1
    instruction_2
        . . .
    instruction_N
end
```

ou bien

```
if (condition)
     ensemble d'instructions 1
else
     ensemble d'instructions 2
end
```

Si la condition est évaluée à **vrai**, les instructions entre le **if** et le **end** seront exécutées, sinon elles ne seront pas (ou si un **else** existe les instructions entre le **else** et le **end** seront exécutées). S'il est nécessaire de vérifier plusieurs conditions au lieu d'une seule, on peut utiliser des clauses **elseif** pour chaque nouvelle condition, et à la fin on peut mettre un **else** dans le cas ou aucune condition n'a été évaluée à **vrai**. Voici donc la syntaxe générale :

```
if (expression_1)
       Ensemble d'instructions 1
elseif (expression_2)
       Ensemble d'instructions 2
              ....
elseif (expression_n)
       Ensemble d'instructions n
else
       Ensemble d'instructions si toutes les expressions étaient fausses
end
```

Par exemple, le programme suivant vous définit selon votre âge :

```
--> age = input('Entrez votre âge : ');...
    if (age < 2)
       disp('Vous êtes un bébé')
    elseif (age < 13)
       disp('Vous êtes un enfant')
    elseif (age < 18)
```

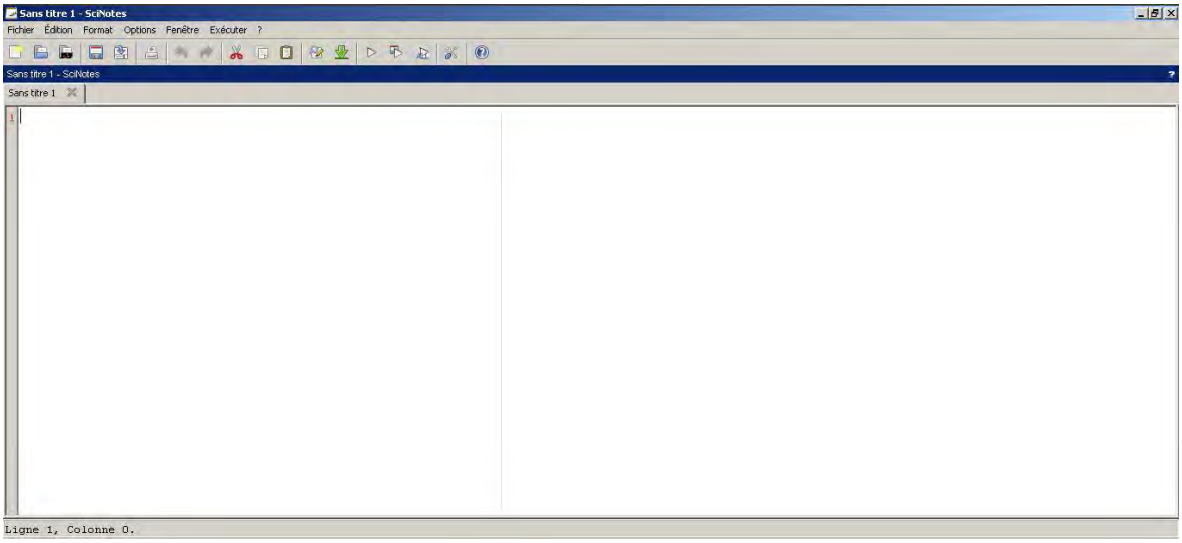
```
    disp ('Vous êtes un adolescent')
elseif (age < 60)
    disp ('Vous êtes un adulte')
else
    disp ('Vous êtes un vieillard')
end
```

Comme vous pouvez le constatez, l'écriture d'un programme SCILAB directement après l'invité de commande (le prompt **-->**) est un peu déplaisant et ennuyeux.

Une méthode plus pratique consiste à écrire le programme dans un fichier séparé, et d'appeler ce programme (au besoin) en tapant le nom du fichier dans l'invité de commande. Cette approche est définie en SCILAB par les **SCI-Files**, qui sont des fichiers pouvant contenir les données, les programmes (scripts) ou les fonctions que nous développons.

Pour créer un **SCI-Files** il suffit de taper la commande **edit**, ou tout simplement aller dans le menu : **Applications → SciNotes** (ou cliquer sur l'icône).

Dans tous les cas une fenêtre d'édition comme celle-ci va apparaitre :



Tout ce qui vous reste à faire et d'écrire votre programme dans cette fenêtre, puis l'enregistrer avec un nom (par exemple : '**Premier_Programme.sce**'). On signale que l'extension des fichiers **SCI-Files** est soit '**.sci**' ou '**.sce**'.

Maintenant, si nous voulons exécuter notre programme, il suffit d'aller à l'invité de commande habituel (**-->**) puis taper le nom de notre fichier avec le '**.sce**' dans la fonction **exec** comme ceci :

```
--> exec('Premier_Programme.sce',0)                    ↵
```

Et l'exécution du programme va démarrer immédiatement.

Pour retourner à la fenêtre d'édition (après l'avoir fermé) il suffit de cliquer sur le fichier et de l'ouvrir avec **Scinotes**.

Exemple :

Créons un programme qui trouve les racines d'une équation de second degré désigné par : $ax^2+bx+c=0$.

Voici le **SCI-File** qui contient le programme, il est enregistré avec le nom : **'Equation2deg.sce'**

```
// Programme de résolution de l'équation a*x^2+b*x+c=0
a = input ('Entrez la valeur de a : '); // lire a
b = input ('Entrez la valeur de b : '); // lire b
c = input ('Entrez la valeur de c : '); // lire c
delta = b^2-4*a*c; // Calculer delta
if delta < 0 // Pas de solution
    disp('Pas de solution')
elseif delta == 0 // Solution double
    disp('Solution double : ')
    x=-b/(2*a)
else // Deux solutions
    disp('Deux solutions distinctes: ')
    x1=(-b+sqrt(delta))/(2*a)
    x2=(-b-sqrt(delta))/(2*a)
end
```

Si nous voulons exécuter le programme, il suffit de taper le nom du programme dans la fonction **exec** :

```
--> exec('Equation2deg.sce',0)        ↵
      Entrez la valeur de a : -2      ↵
      Entrez la valeur de b : 1       ↵
      Entrez la valeur de c : 3       ↵
      Deux solutions :
      x1 =
            -1.
      x2 =
            1.5
```

Ainsi, le programme va être exécuté en suivant les instructions écrites dans son **SCI-File**. Si une instruction est terminée par un point virgule, alors la valeur de la variable concernée ne sera pas affichée, par contre si elle termine par une virgule ou un saut à la ligne, alors les résultats seront affichés.

3.3.2. L'instruction select

L'instruction **select** exécute des groupes d'instructions selon la valeur d'une variable ou d'une expression. Chaque groupe est associé a une clause **case** qui définie si ce groupe doit être exécuté ou pas selon l'égalité de la valeur de ce **case** avec le résultat d'évaluation de l'expression de **select**. Si tous les **case** n'ont pas étés acceptés, il est possible d'ajouter une clause **else** qui sera exécutée seulement si aucun **case** n'est exécuté.

Donc, la forme générale de cette instruction est :

```
select expression
      case valeur_1 then
            Groupe d'instructions 1
      case valeur_2 then
            Groupe d'instructions 2
            . . .
      case valeur_n then
            Groupe d'instructions n
      else
            Groupe d'instructions si tous les case ont échoué
end
```

Exemple :

```
x = input ('Entrez un nombre : ');
select x
       case 0 then
              disp('x = 0 ')
       case 10 then
              disp('x = 10 ')
       case 100 then
              disp('x = 100 ')
       else
              disp('x n''est pas 0 ou 10 ou 100 ')
end
```

L'exécution va donner :

```
Entrez un nombre : 50          ↵
x n'est pas 0 ou 10 ou 100
```

3.3.3. L'instruction for

L'instruction **for** répète l'exécution d'un groupe d'instructions un nombre déterminé de fois. Elle à la forme générale suivante :

```
for variable = expression_vecteur
      Groupe d'instructions
end
```

L'expression_vecteur corresponde à la définition d'un vecteur : début : pas : fin ou début : fin
La variable va parcourir tous les éléments du vecteur défini par l'expression, et pour chacun il va exécuter le groupe d'instructions.

Exemple :

Dans le tableau suivant, nous avons trois formes de l'instruction **for** avec le résultat SCILAB :

L'instruction for	`for i = 1 : 4`
	`  j=i*2 ;`
	`  disp(j)`
	`end`
		`for i = 1 : 2 : 4`
		`  j=i*2 ;`
		`  disp(j)`
		`end`
			`for i = [1,4,7]`
			`  j=i*2 ;`
			`  disp(j)`
			`end`

Le résultat de l'exécution	2. 4. 6. 8.	2. 6.	2. 8. 14.

3.3.4. L'instruction while

L'instruction **while** répète l'exécution d'un groupe d'instructions un nombre indéterminé de fois selon la valeur d'une condition logique. Elle a la forme générale suivante :

```
while (condition)
     Ensemble d'instructions
end
```

Tant que l'expression de **while** est évaluée à **true**, l'ensemble d'instructions s'exécutera en boucle.

Exemple :

```
a=1 ;
while (a~=0)
      a = input ('Entrez un nombre (0 pour terminer) : ') ;
end
```

Ce programme demande à l'utilisateur d'entrer un nombre. Si ce nombre n'est pas égal à **0** alors la boucle se répète, sinon (si la valeur donnée est **0**) alors le programme s'arrête.

3.4. Exercice récapitulatif

Il existe des fonctions prédéfinis en SCILAB donnée dans le tableau ci-dessous. Essayons de les programmer (pour un vecteur donnée **V**).

La fonction	Description	Le programme qui la simule

sum (V)	La somme des éléments d'un vecteur V	`n = length(V);`
		`somme = 0 ;`
		`for i = 1 : n`
		`  somme=somme+V(i) ;`
		`end`
		`disp(somme)`

prod (V)	Le produit des éléments d'un vecteur V	`n = length(V);`
		`produit = 1 ;`
		`for i = 1 : n`
		`  produit=produit*V(i) ;`
		`end`
		`disp(produit)`

mean (V)	La moyenne des éléments d'un vecteur V	`n = length(V);`
		`moyenne = 0 ;`
		`for i = 1 : n`
		`  moyenne = moyenne+V(i) ;`
		`end`
		`moyenne = moyenne / n`

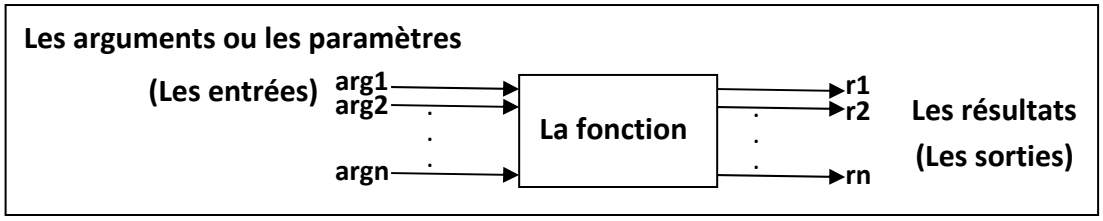
diag (V)	Créer une matrice ayant le vecteur V dans le diagonale, et 0 ailleurs	`n = length(V);`
		`A = zeros(n) ;`
		`for i = 1 : n`
		`  A(i,i)=V(i) ;`
		`end`
		`disp(A)`

gsort (V)	Ordonne les éléments du vecteur V par ordre décroissant	`n = length(V);`
		`for i = 1 : n-1`
		`  for j = i+1 : n`
		`  if V(i) < V(j)`
		`  tmp = V(i) ;`
		`  V(i) = V(j) ;`
		`  V(j) = tmp ;`
		`  end`
		`  end`
		`end`
		`disp(V)`

3.5. Les fonctions

Il existe une différence de concept entre les fonctions en informatique ou en mathématique :

1. En informatique, une fonction est une routine (un sous programme) qui accepte des arguments (des paramètres) et qui renvoie un résultat.



2. En mathématique une fonction **f** est une relation qui attribue à chaque valeur **x** au plus une seule valeur **f(x)**.

3.5.1. Création d'une fonction dans un SCI-Files

SCILAB contient un grand nombre de fonctions prédéfinies comme **sin**, **cos**, **sqrt**, **sum**, …etc. Et il est possible de créer nos propres fonctions en écrivant leurs codes source dans des fichiers **SCI-Files** (portant le même nom de fonction) en respectant la syntaxe suivante :

```
function [r1, r2, …, rn] = nom_fonction (arg1, arg2, …, argn)
      // le corps de la fonction
      . . .
      r1 = . . .  // la valeur retournée pour r1
      r2 = . . .  // la valeur retournée pour
      r2 . . .
      rn = . . .  // la valeur retournée pour rn

endfunction       // le endfunction est facultatif
```

Ou : $r_1 \ldots r_n$ sont les valeurs retournées, et $arg_1 \ldots arg_n$ sont les arguments.

Exemple :
Ecrire une fonction qui calcule la racine carrée d'un nombre par la méthode de Newton (vue dans le TP).
Solution :

```
--> edit racine
```

Le fichier racine.sci

```
function r = racine(nombre)
    r = nombre/2;
    precision = 6;
    for i = 1:precision
        r = (r + nombre ./ r) / 2;
    end
```

L'exécution :

```
--> x = racine(9)
      x =
          3.
--> x = racine(196)
      x =
         14.
--> x = racine([16,144,9,5])
      x =
          4.    12.    3.    2.236068
```

Remarque :

Contrairement à un programme (un script), une fonction peut être utilisée dans une expression par exemple : **2*racine(9)-1.**

3.5.2. Comparaison entre un programme est une fonction

Un programme	**Une fonction**

`a = input('Entrez un nombre positif: ');`
`x = a/2;`
`precision = 6;`
`for i = 1:precision`
`  x = (x + a ./ x) / 2;`
`end`
`disp(x)`
	`function r = racine(nombre)`
	`  r = nombre/2;`
	`  precision = 6;`
	`  for i = 1:precision`
	`  r = (r + nombre ./ r) / 2;`
	`  end`

L'exécution :
`--> exec('racine.sce',0)` ↵
`Entrez un nombre positif: 16` ↵
`4.`
	L'exécution :
	`--> racine(16)` ↵
	`ans =`
	`4.`

on ne peut pas écrire des expressions tel que : `--> 2 * exec('racine.sce',0) + 4` ✗	on peut écrire sans problème des expressions comme : `--> 2 * racine(x) + 4` ✓

4. Les graphiques et la visualisation des données en SCILAB

Partant du principe qu'une image vaut mieux qu'un long discours, SCILAB offre un puissant système de visualisation qui permet la présentation et l'affichage graphique des données d'une manière à la fois efficace et facile.

Dans cette partie du cours, nous allons présenter les principes de base indispensables pour dessiner des courbes en SCILAB.

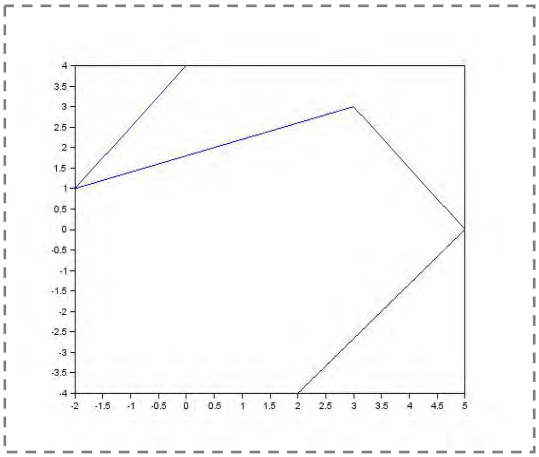
4.1. La fonction plot

La fonction **plot** est utilisable avec des vecteurs ou des matrices. Elle trace des lignes en reliant des points de coordonnées définies dans ses arguments, et elle à plusieurs formes :

- **Si elle contient deux vecteurs de la même taille comme arguments** : elle considère les valeurs du premier vecteur comme les éléments de l'axe X (les abscisses), et les valeurs du deuxième vecteur comme les éléments de l'axe Y (les ordonnées).

Exemple :

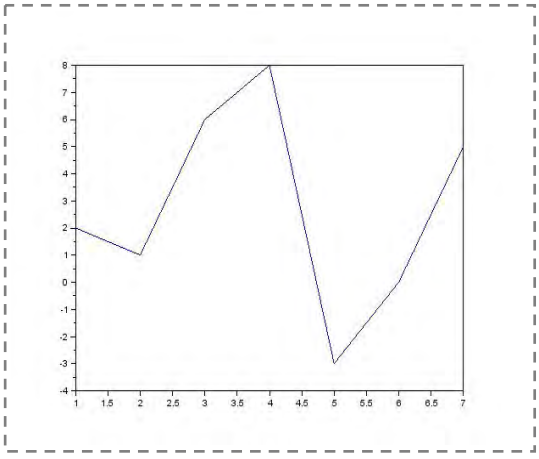
```
--> A = [2, 5, 3,-2,0]
A =
      2.    5.    3.   -2.    0.

--> B = [-4, 0, 3, 1,4]
B =
      -4.   0.    3.    1.    4.

--> plot(A,B)
```



- **Si elle contient un seul vecteur comme argument :** elle considère les valeurs du vecteur comme les éléments de l'axe Y (les ordonnées), et leurs positions relatives définirons l'axe X (les abscisses).

Exemple :

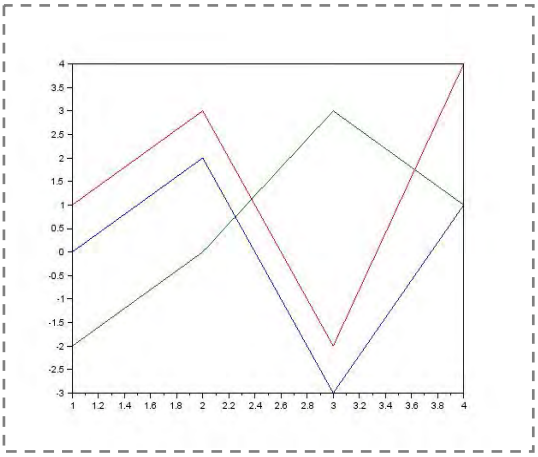
```
--> V = [2, 1, 6, 8,-3, 0,5]
V =
      2.    1.    6.    8.   -3.    0.    5.

--> plot(V)
```



- **Si elle contient une seule matrice comme argument :** elle considère les valeurs de chaque colonne comme les éléments de l'axe Y, et leurs positions relatives (le numéro de ligne) comme les valeurs de l'axe X. Donc, elle donnera plusieurs courbes (une pour chaque colonne).

Exemple :

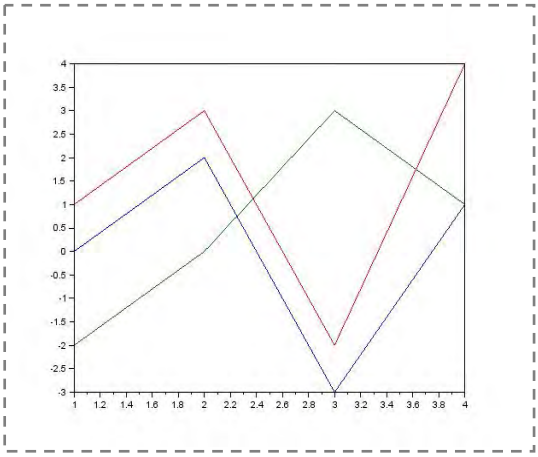
```
--> M = [0 -2 1;2 0 3;-3 3 -2;1 1 4]
M =
    0.    -2.     1.
    2.     0.     3.
   -3.     3.    -2.
    1.     1.     4.

--> plot(M)
```



❖ **Si elle contient deux matrices comme arguments :** elle considère les valeurs de chaque colonne de la première matrice comme les éléments de l'axe X, et les valeurs de chaque colonne de la deuxième matrice comme les valeurs de l'axe Y.

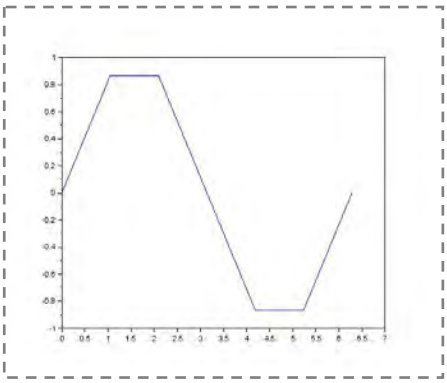
Exemple :

```
--> K = [1 1 1;2 2 2; 3 3 3;4 4 4]
K =
    1.     1.     1.
    2.     2.     2.
    3.     3.     3.
    4.     4.     4.

--> M = [0 -2 1;2 0 3;-3 3 -2;1 1 4]
M =
    0.    -2.     1.
    2.     0.     3.
   -3.     3.    -2.
    1.     1.     4.

--> plot(K,M)
```



Il est évident que plus le nombre de coordonnées augmente plus la courbe devienne précise. Par exemple pour dessiner la courbe de la fonction y = sin(x) sur [0, 2π] on peut écrire :

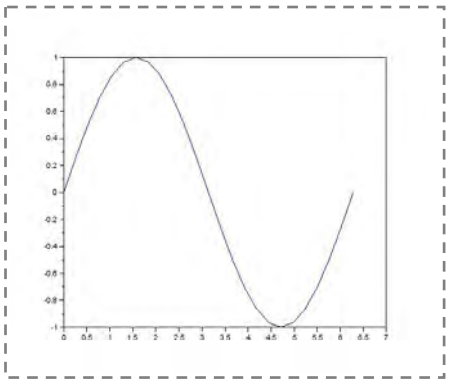
Pas = $\pi/3$



La première figure

```
--> x = 0:%pi/3:2*%pi;
--> y = sin(x);
--> plot(x,y)
```

Pas = $\pi/12$



La deuxième figure

```
--> x = 0:%pi/12:2*%pi;
--> y = sin(x);
--> plot(x,y)
```

4.2. Modifier l'apparence d'une courbe

Il est possible de manipuler l'apparence d'une courbe en modifiant la couleur de la courbe, la forme des points de coordonnées et le type de ligne reliant les points.

Pour cela, on ajoute un nouveau argument (qu'on peut appeler un marqueur) de type chaine de caractère à la fonction **plot** comme ceci :

```
plot (x, y, 'marqueur')
```

Le contenu du marqueur est une combinaison d'un ensemble de caractères spéciaux rassemblés dans le tableau suivant :

Couleur de la courbe		**Représentation des points**	
le caractère	**son effet**	**le caractère**	**son effet**
b ou **blue**	courbe en bleu	**.**	un point **.**
g ou **green**	courbe en vert	**o**	un cercle ●
r ou **red**	courbe en rouge	**x**	le symbole **x**
c ou **cyan**	entre le vert et le bleu	**+**	le symbole **+**
m ou **magenta**	en rouge violacé vif	*****	une étoile *****
y ou **yellow**	courbe en jaune	**s**	un carré ■
k ou **black**	courbe en noir	**d**	un losange ♦
Style de la courbe		**v**	triangle inferieur ▼
le caractère	**son effet**	**^**	triangle supérieur ▲
-	en ligne plein ———	**<**	triangle gauche ◀
:	en pointillé ·····	**>**	triangle droit ▶
-.	en point tiret – · – ·	**p**	pentagramme ★
--	en tiret – – –	**h**	hexagramme ✶

Exemple :

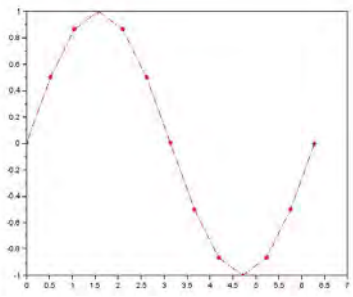
Essayons de dessiner la fonction y = sin(x) pour x = [0 ... 2π] avec un pas = π/6.

```
--> x = 0:%pi/6:2*%pi;
--> y = sin(x);
```

En changeant le marqueur on obtient des résultats différents, et voici quelques exemples :

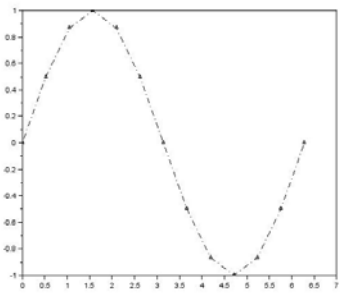
Couleur rouge, en pointillé et avec des étoiles

```
plot(x, y,'r:*')
```



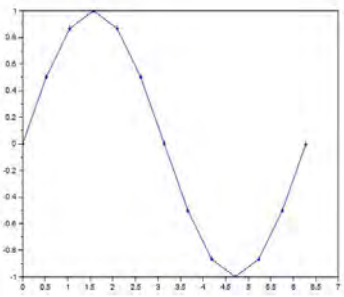
Couleur noire, en point tiret et avec des rectangles sup

```
plot(x, y,'black-.^')
```



Couleur bleu, en ligne plein et avec des pentagrammes

```
plot(x, y,'pb-')
```



4.3. Annotation d'une figure

Dans une figure, il est préférable de mettre une description textuelle aidant l'utilisateur à comprendre la signification des axes et de connaitre le but ou l'intérêt de la visualisation concernée.

Il est très intéressant également de pouvoir signaler des emplacements ou des points significatifs dans une figure par un commentaire signalant leurs importances.

- ✓ Pour donner un titre à une figure contenant une courbe nous utilisons la fonction **title** comme ceci :

```
--> title('titre de la figure')
```

- ✓ Pour donner un titre pour l'axe vertical des ordonnées y, nous utilisons la fonction **ylabel** comme ceci :

```
--> ylabel('Ceci est l''axe des ordonnées Y')
```

- ✓ Pour donner un titre pour l'axe horizontal des abscisses x, nous utilisons la fonction **xlabel** comme ceci :

```
--> xlabel('Ceci est l''axe des abscisses X')
```

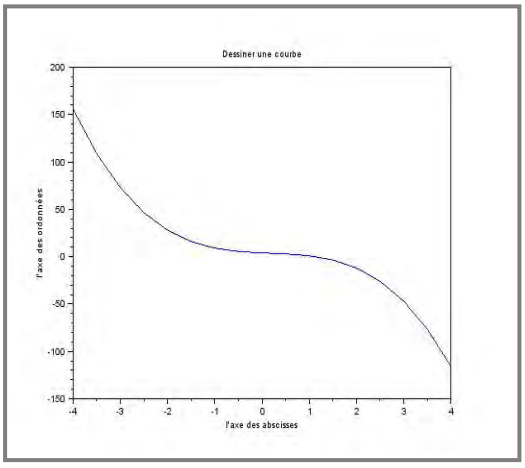
- ✓ Pour écrire un texte (un message) sur la fenêtre graphique à une position indiquée par les coordonnées **x** et **y**, nous utilisons la fonction **xstring** comme ceci :

```
--> xstring(x, y, 'Ce point est important')
```

Exemple :

Dessinons la fonction : $y = -2x^3+x^2-2x+4$ pour x varie de -4 jusqu'à 4, avec un pas = 0.5.

```
--> x = -4:0.5:4;
--> y = -2*x^3+x^2-2*x+4;
--> plot(x,y)
--> title('Dessiner une courbe')
--> xlabel('l''axe des abscisses')
--> ylabel('l''axe des ordonnées')
```



4.4. Dessiner plusieurs courbes dans la même figure

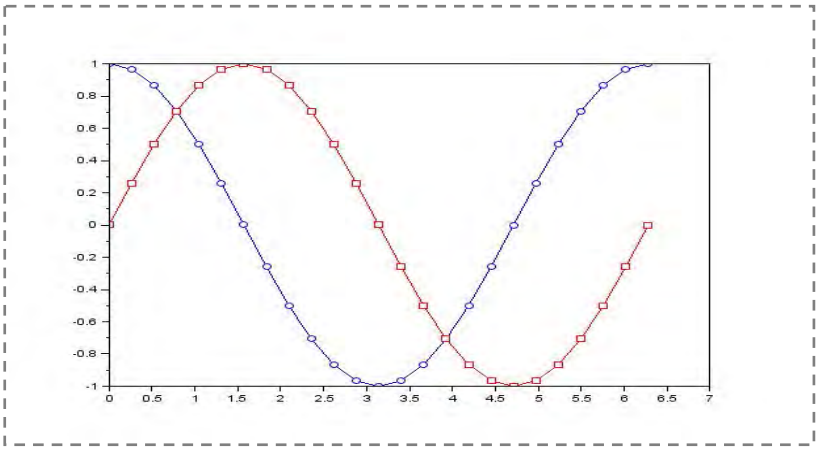
Par défaut en SCILAB, chaque nouveau dessin avec la commande **plot** efface le précédent. Pour forcer une nouvelle courbe à coexister avec les précédentes courbes, Il existe au moins trois méthodes :

4.4.1. La commande mtlb_hold

La commande **mtlb_hold** (ou **mtlb_hold on**) active le mode 'préservation des anciennes courbes' ce qui permette l'affichage de plusieurs courbes dans la même figure. Pour annuler son effet il suffit de réécrire **mtlb_hold** (ou **mtlb_hold off**).

Par exemple pour dessiner la courbe des deux fonctions cos(x) et sin(x) dans la même figure, on peut écrire :

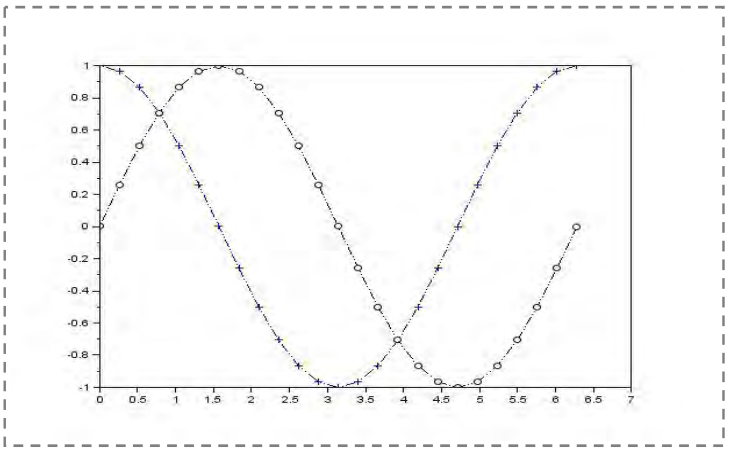
```
--> x=0:%pi/12:2*%pi;
--> y1=cos(x);
--> y2=sin(x);
--> plot(x,y1,'b-o')
--> mtlb_hold on
--> plot(x,y2,'r-s')
```



4.4.2. Utiliser plot avec plusieurs arguments

On peut utiliser **plot** avec plusieurs couples (x,y) ou triples (x ,y, 'marqueur') comme arguments. Par exemple pour dessiner les mêmes fonctions précédentes on écrit :

```
--> x=0:%pi/12:2*%pi;
--> y1=cos(x);
--> y2=sin(x);
--> plot(x,y1,'b:+',x,y2,'k:o')
```

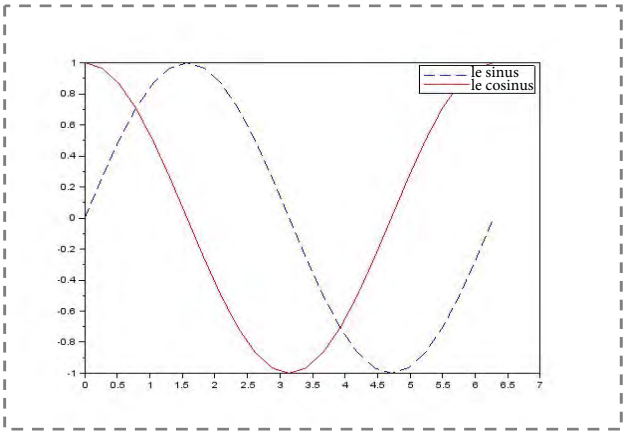


4.4.3. Utiliser des matrices comme argument de la fonction plot

Dans ce cas on obtient plusieurs courbes automatiquement pour chaque colonne (ou parfois les lignes) de la matrice. On a déjà présenté ce cas plus auparavant.

Après pouvoir parvenir à mettre plusieurs courbes dans la même figure, il est possible de les distinguer en mettant une légende indiquant les noms des ces courbes. Pour cela, on utilise la fonction **legend**, comme illustre l'exemple suivant qui dessine les courbes des deux fonctions sin(x) et cos(x) :

```
--> x=0:%pi/12:2*%pi;
--> y1=sin(x);
--> y2=cos(x);
--> plot(x,y1,'b--',x,y2,'-r')
--> legend('le sinus','le cosinus')
```



Il est possible de déplacer la légende (qui se situe par défaut dans le coin supérieur droit) en utilisant la souris avec un glisser-déposer.

4.5. Manipulation des axes d'une figure

SCILAB calcule par défaut les limites (le minimum et le maximum) des axes X et Y et choisie automatiquement le partitionnement adéquat. Mais il est possible de contrôler l'aspect des axes via la commande **mtlb_axis**. Pour définir les limites des axes il est possible d'utiliser cette commande avec la syntaxe suivante :

mtlb_axis ([xmin xmax ymin ymax]) Ou **mtlb_axis ([xmin,xmax,ymin,ymax])**

Avec : **xmin** et **xmax** définissent le minimum et le maximum pour l'axe des abscisses.

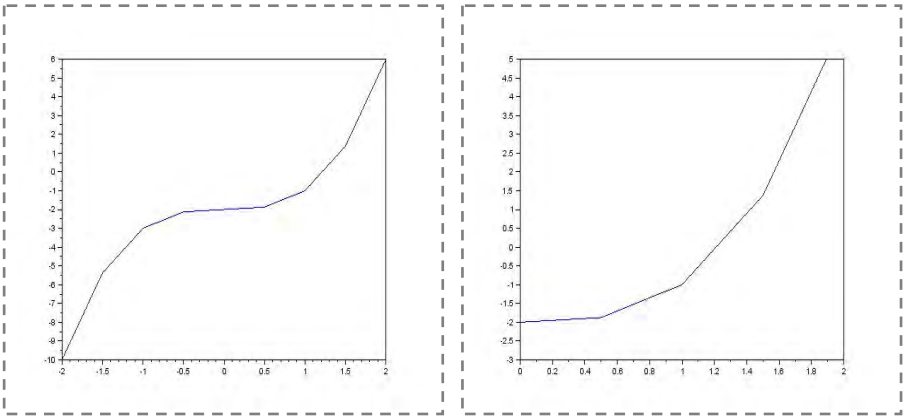
ymin et **ymax** définissent le minimum et le maximum pour l'axe des ordonnées.

Pour revenir au mode d'affichage par défaut, nous écrivons la commande : **mtlb_axis auto**

Exemple :

$f(x) = x^3-2$

```
--> x = -2:0.5:2;
--> y = x.^3-2;
--> plot(x,y)
```



```
--> mtlb_axis auto
```

```
--> mtlb_axis([0,2,-3,5])
```

Parmi les autres options de la commande **mtlb_axis**, nous présentons les suivantes :

– Pour rendre la taille des deux axes identique (la taille et non le partitionnement, nous utilisons la commande **mtlb_axis square**. Elle est nommée ainsi car elle rend l'aspect des axes tel un carré.

– Pour rendre le partitionnement des deux axes identique nous utilisons la commande **mtlb_axis equal**.

– Pour revenir à l'affichage par défaut et annuler les modifications nous utilisons la commande **mtlb_axis auto**.

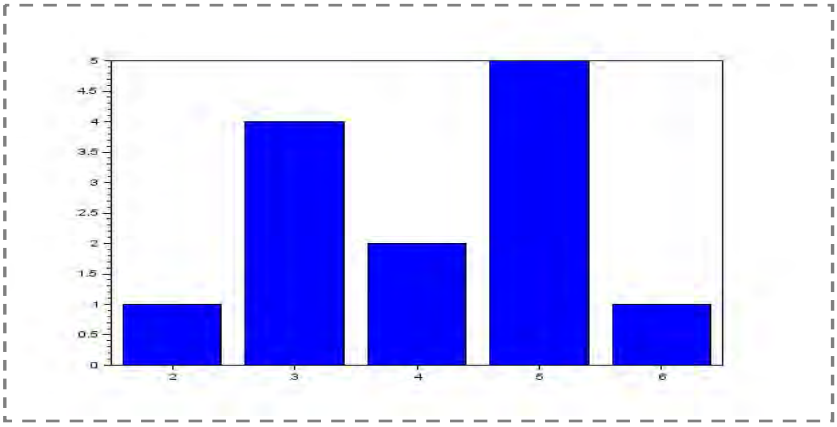
– Pour rendre les axes invisibles nous utilisons la commande **mtlb_axis off**. Pour les rendre visibles à nouveau nous utilisons la commande **mtlb_axis on**.

4.6. D'autres types de graphiques

Le langage SCILAB ne permet pas uniquement l'affichage des points pour tracer des courbes, mais il offre aussi la possibilité de tracer des graphes à bâtons et des histogrammes. Pour tracer un graphe à bâtons nous utilisons la fonction **bar** qui a le même principe de fonctionnement que la fonction **plot**.

Exemple :

```
--> X=[2,3,5,4,6];
--> Y=[1,4,5,2,1];
--> bar(X,Y)
```



Il est possible de modifier l'apparence des bâtons, et il existe la fonction **barh** qui dessine les bâtons horizontalement, et la fonction **bar3** qui ajoute un effet 3D.

Parmi les fonctions de dessins très intéressantes non présentées (faute de place) on peut trouver : **hist**, **stairs**, **stem**, **pie**, **pie3**, ...etc. (que nous vous encourageons à les étudier).

Nous signalons aussi que SCILAB permet l'utilisation d'un système de coordonnées autre que le système cartésien comme le système de coordonnées polaire (pour plus de détail chercher les fonctions **compass**, **polar** et **rose**).