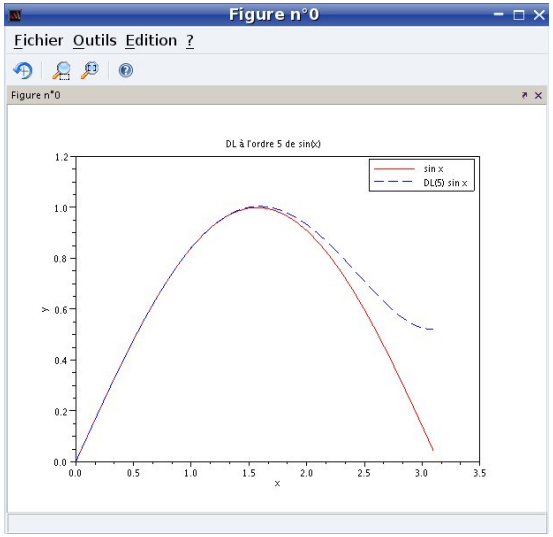
Petit guide de démarrage en scilab

Richard Wilson & Maria Barbi



Licence de Physique à Distance

Table des matières

Chapitre 1 Introduction

Ce petit guide a pour but de vous aider à démarrer sous scilab : comprendre ce qu'est scilab, apprendre quelques commandes, tracer une courbe, écrire vos premiers programmes. Vous y trouverez des nombreux exemples simples : il est vivement conseillé de les exécuter sous scilab, d'abord à l'identique puis en les modifiant, jusqu'à être sûr d'avoir bien integré les notions présentées.

1.1 Qu'est ce que scilab ?

Scilab est un logiciel comprenant à la fois un langage de programmation et une interface utilisateur. Le langage de programmation est constitué d'une riche collection de fonctions (de commandes) permettent de calculer mais aussi de tracer des graphiques. L'interface utilisateur permet d'interpréter les commandes dans une fenêtre, et comprend nombre d'outils utiles : aide en ligne, création d'image à partir des graphiques, etc.

Les commandes scilab sont interprétées – il n'y a donc pas de compilation – ce qui rend les programmes faciles à écrire et à mettre au point.

Scilab est constitué de fonctions couvrant de nombreux domaines du calcul scientifique, par exemple :
- Algèbre linéaire
- Interpolation, approximation ;
- Solveur d'équation différentielles ;
- Traitement du signal ;
- Probabilité et statistiques.

Scilab comprend aussi des fonctions graphiques (2D et 3D), des fonctions permettant de construire des interfaces graphiques ainsi qu'un environnement de simulation et modélisation (Xcos).

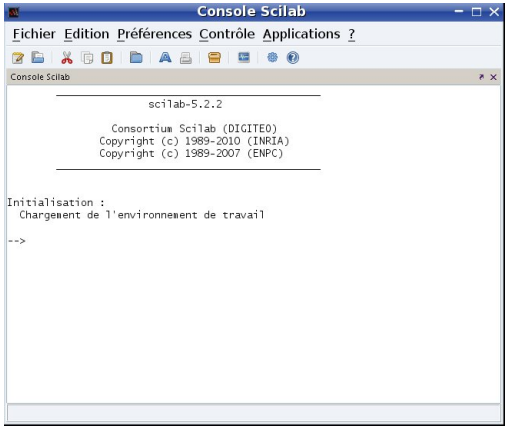
Enfin, scilab est un logiciel libre, donc gratuit, existant pour plusieurs plateformes (Windows, Mac, Linux).

1.1.1 Installer scilab

L'installation des versions binaires (compilées) est aisé sur toutes les machines : http://www.scilab.org/products/scilab/download.

1.2 Premiers pas

Lorsque vous lancez scilab (soit en cliquant sur l'icône correspondante, soit en exécutant la commande `scilab` dans un terminal, selon les environnements), une fenêtre s'ouvre.
Dans cette fenêtre, vous ne pourrez entrer que des commandes scilab ou lancer des programmes écrits en scilab.
Après la saisie d'une commande, appuyez sur la touche «Return» ou «Entrée». La commande est alors interprétée et exécutée.
Pour rappeler la commande précédente (pour la modifier ou la corriger), appuyer sur la flèche «déplacement vers le haut». Il est ainsi possible de remonter aux commandes saisies précédemment.



Une fois une commande rappelée, vous pouvez la modifier puis la ré-exécuter en appuyant sur «Entrée».

Exemple 1.1 :

```
-> a=3
a  =

    3.
--> a+2*5
ans  =

    9.

-->
```

La commande `a=3` affecte la valeur `3` à la variable `a`, le résultat s'affiche après exécution. La variable `a` est réutilisable pour des calculs ultérieurs. Si le résultat d'un calcul n'est pas affecté à une variable particulière, comme dans la commande `a + 2*3`, scilab stocke la réponse dans une variable nommé `ans` (answer = réponse).

1.2.1 Aide en ligne : `help`

Une aide en ligne est disponible. Taper : `help` (suivi de «Entrée»)

Une fenêtre s'ouvre alors avec une liste des rubriques.

Pour une aide sur une commande précise, taper `help nom-de-la-commande`, par expemple : `help log10`

1.2.2 Répertoire courant

Un point important est de comprendre qu'en ouvrant une fenêtre scilab, vous êtes situé dans un certain répertoire, appelons le, le «répertoire courant». La commande

```
--> ls
```

affiche le contenu du répertoire courant, c.à.d. le nom des fichiers et répertoires s'y trouvant.

Pour connaître le nom du répertoire courant, taper :

```
--> pwd
```

S'affiche le nom du répertoire.

Nous reviendrons plus loin sur le commandes permettant de changer de répertoire courant.

1.2.3 Méthodes de travail

Il y a deux moyens d'exécuter des commandes scilab.

1. Sasir les commandes l'une après l'autre à la suite du prompt `-->`.

   Cette pratique peut s'avérer pratique pour de petits calculs, ou pour visualiser des données. Cependant, un tel usage, pas différent finalement de l'usage d'une calculatrice, est vite limité lorsqu'on veut exécuter des tâches complexes.

2. Écrire une suite de commande dans un fichier, sauvegarder puis exécuter le fichier (voir aussi section suivante). Le fichier contiendra une suite de commande scilab, généralement une par ligne. Supposons que le fichier soit enregistré sous le nom «`toto.sce`». Il est exécuté (interprété) par scilab par la commande :

   ```
   --> exec toto.sce
   ```

Un fichier contenant des commandes scilab s'appelle un script.

Les avantages de la seconde méthodes sont évidents :
– pas besoin de ressaisir des commandes répétitives ;
– conservation des programmes dans des fichiers, qui sont réutilisables et modifiables ;
– mise au point de tâches longues et complexes ;
– possibilité de créer ses propres fonctions[1] (nous verrons plus loin comment le faire).

[1] Une fonction est une tâche élémentaire qui est exécutée comme une commande. Par exemple, `sin` ou `log` sont des fonctions. Un script est un fichier contenant une succession de commande incluant des appels à des fonctions.

1.2.4 L'éditeur

Pour écrire dans un fichier, n'importe quel éditeur de texte fait l'affaire[a]. Scilab possède un éditeur : **menu application/Editeur**, ou taper simplement :

```
--> edit
```

A droite, le fenêtre de l'éditeur. Une fois le code du programme saisi, enregistrer le dans un fichier : menu `file > Enregistrer`. Il existe aussi une icône dans le bandeau d'outils de l'éditeur scilab.

[a] Attention, les programmes de *traitement* de texte (Word, Openoffice, etc...) ne sont pas des *éditeurs* de texte, ces programmes ajoutant des caractères non-visibles de mise en forme.

La fenêtre de l'éditeur scilab.

1.3 Un premier programme

Un premier exemple. Saisir le code

```
printf("Hello tout le monde ! ")
```

dans l'éditeur. Puis sauver le fichier **dans le répertoire courant** sous le nom «hello.sce» à l'aide du menu **fichier/Enregistrer sous**.

Pour exécuter le script, taper :

```
--> exec hello.sce
```

Un premier programme.

Vous remarquez que les lignes précédés de «//» sont des commentaires et ne sont pas interprétés. Le signe «//» peut aussi être placé au milieu d'une ligne : dans ce cas, tout ce qui suit n'est pas interprété.

Si vous n'avez pas fait d'erreur de frappe, et **si le fichier est bien dans le répertoire courant** (source d'erreur fréquente), voici le résultat ce que cela donne.

```
--> exec Hello.sce

--> // un premier programme

--> printf("Hello tout le monde ! ")
Hello tout le monde !

-->
```

 scilab distingue majuscule et minuscule. La commande `exec hello.sce` renvoie une erreur, le fichier «hello.sce» n'existant pas. Il en est de même pour les noms de variables : `x` et `X` sont deux variables distinctes.

Si vous ne souhaitez pas voir s'afficher les lignes de commande et de commentaires contenues dans le fichier, ce qui peut vite être fastidieux et inutile, il faut rajouter la commande `mode(-1)` au début du script. Ne s'affiche alors que les résultats de commandes d'affichage et non les commandes elle-mêmes.

```
mode(-1)
// un premier programme

printf("Hello tout le monde ! ")
```

La fenêtre de l'éditeur.

Après avoir sauvegardé le fichier, relancer le programme :

```
--> exec Hello.sce

--> mode(-1)
Hello tout le monde !

-->
```

Chapitre 2 Calcul matriciel

Les matrices sont les objets de base que l'on manipule, transforme, ou trace avec scilab. Les matrices sont des tableaux de nombres qui sont, soit le résultat de calculs, soit des données expérimentales. La taille d'une matrice est caractérisée par ses dimensions : nombre de lignes et de colonnes. Un vecteur est une matrice dont l'une des dimensions est 1.

2.1 Initialiser un vecteur ou une matrice

Une matrice (un vecteur) peut être créé en spécifiant un par un leurs coefficients entre crochet :

```
--> v = [ 10 20 30 ]; // vecteur ligne

--> v = [ 10 20 30 ]'; // vecteur colonne («'» = opérateur de transposition)

--> m = [ 0.1 0.2 0.3; 1 2 3; 10 20 30 ] // matrice carrée 3x3
 m  =

    0.1    0.2    0.3
    1.     2.     3.
    10.    20.    30.
```

Le signe «;» a deux significations

1. Utilisé à la finc d'une commande, le «;» rend celle-ci muette : le résultat n'est pas affiché. Si vous souhaitez voir s'afficher le résultat d'une commande, il suffir d'omettre le «;» à la fin.
2. Utilisé entre à l'intérieur des crochets définissant une matrice, le «;» signifie «passer à la ligne».

Remarque : Une fois créée, une variable est stockée en mémoire et donc réutilisable pour un calcul ultérieur. Par exemple, pour voir le contenu de la variable `v` précédemment définie, il suffit de taper : `v` (sans «;»).

Chaque redéfinition du vecteur `v` efface de la mémoire la définition précédente.

Pour créer un vecteur de grande taille dont les coefficients suivent une progression arithmétique, on écrira :

```
--> v = 10:100;          // v = [ 10 11 12 ... 100 ]

--> v = [ 10:0.5:20 ] // v = [ 10 10.5 11 11.5 12 ... 20 ]
 v  =
          column  1 to 15

    10.    10.5    11.    11.5    12.    12.5    13.    13.5    14.    14.5
    15.    15.5    16.    16.5    17.

         column 16 to 21

    17.5    18.    18.5    19.    19.5    20.

--> v = linspace(1, 2, 10); // 10 valeurs équiréparties entre 1 et 2
```

Noter l'utilisation de la commande `linspace`, permettant de construire un vecteur d'éléments qui progressent linéairement entre une valeur initiale et une valeur finale. De manière similaire, la commande `logspace(A, B, N)` permet de définir un vecteur contenant `N` valeurs entre `A` et `B` repartis selon une progression logaritmique.

Les fonctions `ones` ou `zeros` permettent de créer un vecteur ou une matrice dont tous les coefficients sont identiques :

```
-->  v   =   zeros(1, 100); // vecteur de 100 zéros, 1 ligne, 100 colonnes

-->  m   =   ones(3, 3);    // matrice 3x3 remplie de uns

-->  m   =   10*ones(2, 3); // matrice 2x3 remplie de 10

-->  m   =   zeros(2, 3) + 10; // idem
```

2.1.1 Connaître les dimensions d'un vecteur ou d'une matrice

La fonction `length` retourne le nombre d'éléments d'une matrice :


```
--> nm = length(m)
 nm  =

    6.

--> nv = length(v)
 nv  =

    100.
```

La fonction `size` renvoie un vecteur de deux valeurs contenant le nombre de lignes et le nombre de colonnes de la matrice ou du vecteur fourni en argument :

```
--> m = zeros(2, 3);     // matrice 2 lignes, 3 colones

--> size(m)              // affiche 2  3
ans  =

    2.    3.

--> [nl nc] = size(m)    // nl vaut 2, nc vaut 3 (nl et nc sont des variables)
nc  =
    3.
 nl  =
    2.
```

Assigner les dimensions d'une matrice à des variables, `nl` et `nc` dans l'exemple précédent, est souvent utile pour la programmation.

La fonction `size` est également utile pour déterminer si un vecteur est stocké sous la forme d'une ligne ou d'une colonne :

```
-->   v = zeros(1, 5); // vecteur ligne de 5 zéros

-->   length(v)        // affiche 5

-->   size(v)          // affiche 1  5

-->   vt = v';         // transposition du vecteur v
```

```
-->   length(vt)       // affiche 5

-->   size(vt)         // affiche 5  1
```

2.1.2 Accéder aux éléments d'un vecteur ou d'une matrice

Un élément d'un vecteur est repéré par un indice de position. Un élément d'une matrice est repéré par deux indices de position. L'accès à un élément permet de le lire ou d'en changer la valeur (affectation) :

```
-->   v(1) // affiche la valeur du premier coefficient du vecteur
ans  =

    0.

-->   m(2, 3) // affiche le coeff. de la deuxième ligne, troisième colonne
ans  =

    0.

-->   v(1) = 7; // affectation de la valeur 7 au premier coefficient

-->   m(2, 3) = m(2, 3) + 1 // ajoute 1 à m(2,3)
 m  =

    0.    0.    0.
    0.    0.    1.
```

Le symbole « :» permet de spécifier des des blocs d'indices. Deux utilisations possibles :
– L'écriture «i : j» signifie "tous les indices de i à j".
– L'écriture « :» (sans indice) signifie "tous les éléments de la ligne ou la colonne".

```
-->   v(1:3)          // affiche v(1)  v(2)  v(3)
ans  =

    0.    0.    0.
```



```
-->   v(2:4) = 1      // affectation v(2)=v(3)=v(4)=1
 v  =

    0.    1.    1.    1.    0.

-->   b = m(1:2,1:2); // extraction du block 2x2 supérieur gauche

-->   m(1,:) = 0;     // mise à zéro de toute la première ligne

-->   v = m(:,2)      // extraction de toute la deuxième colonne
v  =

    0.
    0.
```

2.2 Opérations élémentaires sur les matrices et les vecteurs

2.2.1 Opérateurs matriciels

Les opérateurs élémentaires `+`, `-`, `*`, `/` et `^` (puissance) de scilab sont des opérateurs matriciels. Cela signifie par exemple que vous ne pouvez additionner ou soustraire que des matrice de mêmes dimension. Avec une exception toutefois : il est possible d'ajouter un nombre (un scalaire) à une matrice ou un vecteur.

De même, vous ne pouvez multiplier deux matrices que si le nombre de colonnes de la première est égale au nombre de lignes de la seconde.

Exemple 2.1 :

```
--> A = [1 2 3 ; 4 5 6];

--> B = [7 8 ; 9 10 ; 11 12];

--> A*B
 ans  =

    58.     64.
    139.    154.

--> A+1        // Ajout d'un scalaire à une matrice
ans  =
```

```
    2.    3.    4.
    5.    6.    7.

--> A+B
    !--error 8
Addition incohérente.

--> A+B'  // B' est la transposée de B
 ans  =

     8.    11.    14.
    12.    15.    18.

--> A^2
    !--error 20
Mauvais type pour le premier argument: une matrice carrée est attendue.
```

 Les opérateurs `*` et `^` étant des opérateurs matriciels, `A^2` signifie `A*A` et retourne ici une erreur (`A` n'étant pas carrée).

2.2.2 Opérateurs «terme à terme»

Il existe toutefois une manière de calculer le carré de tous les éléments d'un vecteur un d'une matrice, ou de les multiplier par les éléments d'une autre matrice. Les opérateurs `.*` et `.^` sont des opérateurs «terme à terme». Ainsi `A.^2` retourne une matrice de même dimension que `A` dont les éléments sont ceux de `A` au carré.

Exemple 2.2 :

```
--> A.^2    // Mise au carré des éléments de A
 ans  =
     1.     4.     9.
    16.    25.    36.

--> A.*B
     !--error 9999 \\
inconsistent element-wise operation

--> A.*B'  // A*transposé(B)
```



```
 ans  =
     7.    18.    33.
    32.    50.    72.
```

2.2.3 Fonctions élémentaires

Les fonctions élémentaires (`cos, sin, log, exp, ...`) appliquées à une matrice retournent la valeur de la fonction appliquée à chacun des termes de la matrice.

Exemple 2.3 :

```
--> cos(A)
 ans  =
    0.5403023  - 0.4161468  - 0.9899925
  - 0.6536436    0.2836622    0.9601703
```

2.3 La boucle `for`

La boucle `for` sert à répéter une opération sur les composantes d'un vecteur. On parle alors d'«itération».

Voici un exemple simple (simpliste). On souhaite élever au carré les éléments d'un vecteur `X`. On va donc parcourir les indices du tableau du premier au dernier : de `1` à `length(X)`.

Exemple 2.4 :

```
--> X = 1:0.5:10;

--> for i=1:length(X),
-->   X2(i) = X(i)*X(i);
--> end
```

Le vecteur `X2` a même taille que `X` et contient les carrés des valeurs de `X`.

 L'exemple précédent illustre la syntaxe d'une boucle `for`. Il est cependant «idiot» en terme de programmation car les opérateurs matriciels `.*` ou `.^` permettent d'effectuer la même opération de façon beaucoup plus efficace :

```
--> X2 = X.^2;  // X2 : éléments de X au carré

--> X2 = X.*X;  // la même chose
```

Conclusion : Les opérateurs matriciels permettent d'éviter la plupart des boucles `for`. En outre, ils sont beaucoup plus efficaces (plus rapides). Les boucles `for` sont donc à éviter autant que faire se peut, ce qui n'est pas toujours possible.

Remarque : Il est donc possible de construire une matrice à partir de n'importe quelle fonction appliquéé à `X`. Par exemple :

```
--> XF = X.^3 .* cos(X);  // XF = fonction x^2 cos(x) appliquée à X
```

2.4 Le test `if`

La commande `if`[1] permet de tester une condition. La ou les commande(s) suivant le test, et précédent le `end`, sont effectuées si la condition est réalisée.

Exemple 2.5 : On souhaite sommer les termes strictement supérieurs à 0.5 d'un vecteur aléatoire.

```
--> X = rand(1000,1);

---> som = 0;

--> for i=1:length(X),
-->   if X(i) > 0.5, som = som + X(i); end
--> end

--> disp(som)

    165.4445
```

Remarque : La fonction `rand(nl,nc)` permet de générer une matrice `nl`×`nc` de nombres aléatoires suivant une loi uniforme ou une loi normale. Il existe une autre fonction permettant de générer des nombres aléatoires suivant d'autres lois : `grand`.

[1] Comme la boucle `for`, le test `if` se termine par un `end`

2.5 Opérateurs logiques et relationnels

Les conditions utilisées dans les boucle `if` sont souvent construite sur la base d'opérateurs relationnels (supérieur, inférieur, différent de...) et peuvent être combinées grâce à des opérateurs logiques (si la *condition 1* et/ou la *condition 1* sont réalisées, alors...). Voici la syntaxe utilisée pour ces opérateurs dans scilab.

Opérateurs logiques	instruction scilab
et	&
ou	|
non	∼

Les opérateurs logiques.

Relations	instruction scilab
$=$	=
$\neq$	∼= ou <>
$<$	<
$>$	>
$\leqslant$	<=
$\geqslant$	>=

Les opérateurs relationnels.

2.6 Variables spéciales et nombres complexes

Le nombre complexe $\sqrt{-1}$ est prédefini dans scilab et stocké dans la variable spéciale `%i`. Cette variable commence par le symbole `%` pour indiquer qu'elle est prédefinie et ne peut pas être modifiée. Voir aussi le contenu de la variable `%pi`.

Pour entrer une variable complexe, il suffit de taper par exemple
`--> z1 = 1 + 2.5*%i`
ou
`--> z2 = 0.4 - %i`
(`%i` étant une variable, il faut utiliser le signe de multiplication). Les variables complexes peuvent être manipulées comme les variables réelles (taper par exemple `z1+z2` ou `z2/2`). Une varible complexe dont la partie imaginaire est nulle est traitée comme une variable réelle.

Scilab contient aussi des fonctions prédéfinies utiles pour manipuler les complexes :

```
--> real(z1)   // partie réelle
ans =
      1.

---> imag(z1) // partie imaginaire
ans =
      2.5

--> modul = abs(z1)  // module du nombre complexe
ans  =
```

```
    2.6925824

--> theta = imag(log(z1)) // calcul de l'argument du nombre complexe
ans  =

    1.1902899

--> modul*exp(%i*theta)
ans  =

    1. + 2.5i
```

La dernière commande montre comment retrouver le nombre complexe original à partir de son module et de sa phase.

Remarque : Les éléments d'une matrice peuvent bien sûr être complexes. Les opérateurs décrites ci-dessus s'appliquent alors à chaque élément de la matrice complexe.

Chapitre 3 Répertoires et fichiers

3.1 Se localiser dans le système de fichiers

Sous Linux, Le répertoire courant est le répertoire du shell depuis lequel scilab a été lancé. Sous Windows ou Mac, cela dépend de l'installation de scilab. Pour connaître le répertoire courant taper : `pwd`.

Il est possible de changer à tout moment de répertoire courant grâce à la commande `cd` (change directory), qui prend comme argument le nom du répertoire dans lequel on souhaite se placer.

```
--> cd 'nom_de_répertoire'
```

(il faut bien sûr que le répertoire «`nom_de_répertoire`» existe, sinon, vous n'obtiendrez qu'un message d'erreur.)

Les règles d'adressage (absolu, relatif) sont celles d'Unix. Ainsi, deux points (`..`) indiquent toujours le répertoire racine du repertoire courant, et
`--> cd '..'`
permet de remonter d'un cran dans l'arborescence des répertoires,
`--> cd '../..'`
de deux crans, etc.
Pour connaître à tout moment le nom du répertoire courant, utiliser la commande `pwd` :
La commande `cd` sans argument définit le répertoire racine comme le répertoire courant.

```
--> pwd
 ans  =
 /home/riw/documents/ens/maths/lp341

--> cd '..'
 ans  =
 /home/riw/documents/ens/maths

-->pwd
 ans  =
 /home/riw/documents/ens/maths

--> cd '../..'
 ans  =
 /home/riw/documents
```

Il existe aussi (au moins sous Linux) des items du menu `Fichiers` de la fenêtre de commande permettant de faire ces opérations.

menu `Fichier > Affichier le répertoire courant`

menu `Fichier > Changer le répertoire courant`

3.2 Écrire ou lire un vecteur ou une matrice dans un fichier

Il est souvent intéressant de conserver un vecteur ou une matrice dans un fichier afin de pouvoir le réutiliser lors d'une autre session.

3.2.1 Ecrire un vecteur dans un fichier

Pour enregistrer un vecteur ou une matrice dans un fichier au format ASCII (fichier texte), utiliser la fonction `write`.

Un vecteur colonne est enregistré sous la forme d'une colonne de valeurs dans le fichier. Suivant sa dimension, un vecteur ligne est enregistré sous la forme d'une ou plusieurs lignes.


Une erreur est générée si le fichier spécifié existe déjà.

```
--> x = rand(10,1) ; // x vecteur colonne de 10 éléments tirés au hasard

--> write('data1',x); // fichier 'data1' créé dans le répertoire courant
                      //  contenant ces données

--> m = rand(10,3) ; // m matrice 10 lignes 3 colonnes

--> write('data2',m);

--> write('data1',m);
                 !--error 240
Le fichier "data1" existe déjà ou le répertoire n'est pas accessible en écriture.
```

Pour supprimer le fichier `data1`, sous Linux ou Mac, taper :

```
--> unix('rm data1');
```

sous Windows, taper :

```
--> unix('del data1');
```


3.2.2 Lire un vecteur ou une matrice dans un fichier

La fonction `read` permet de lire des matrices stockées dans un fichier texte (format ASCII). La matrice lue peut être affectée à une variable. Le fichier est supposé contenir une ou plusieurs colonnes de nombres. Il est possible de sélectionner le nombre de lignes et de colonnes lues (deuxième et troisième arguments de la commande `read`) :

```
--> x = read('data1',5,1) // lit les 5 premières ligne de la première colonne
 x  =

    0.4094825
    0.8784126
    0.1138360
    0.1998338
    0.5618661

--> q = read('data',-1, 2) // lit entièrement les deux premières colonnes
q  =

    0.3873779    0.5376230
    0.9222899    0.1199926
    0.9488184    0.2256303
    0.3435337    0.6274093
    0.3760119    0.7608433
    0.7340941    0.0485566
    0.2615761    0.6723950
    0.4993494    0.2017173
    0.2638578    0.3911574
    0.5253563    0.8300317
```

Fixer à `-1` le nombre de lignes permet de lire toutes les lignes (même si on en connaît pas le nombre à priori).

3.2.3 Lire un fichier contenant des caractères

Pour une lire une séquence de caractères dans un fichier (il pourrait s'agir par exemple une séquence d'ADN) ajouter un quatrième argument, "(a)", à la fonction `read` indiquant que les données sont des caractères.

Dans le premier exemple ci-dessous, la chaîne `s1` est lue dans un fichier contenant la séquence ADN sur une seule ligne. Dans le second exemple, la chaîne `s2` est écrite sur plusieurs lignes dans le fichier `data2`. Après lecture du fichier, les lignes sont stockées dans la variable `v`. La chaîne `s2` est ensuite formée en «collant» à la suite les lignes de `v` (concaténation).

```
--> s1 = read('data1', 1, 1, "(a)" );


--> v = read('data2', -1, 1, "(a)" ); // v vecteur de chaînes de caractères

--> s2 = strcat(v); // concaténation des chaînes de caractères
```

3.3 Sauver et charger un environnement

Pour enregistrer toutes les variables et fonctions qui ont été créés lors d'une session («l'environnement») :

menu : `fichier > Enregistrer l'environnement...` puis saisir un nom de fichier avec l'extension `.sav`.

ou taper la commande

```
save('nom-de-fichier.sav');
```

Pour charger l'environnement lors d'une autre session :

menu `Fichier -> Charger l'environnement`

ou taper la commande :

```
load('nom-de-fichier.sav');
```

Chapitre 4 Graphisme

4.1 La fonction `plot`

La fonction à utiliser pour la plupart des graphiques 2D est `plot`. Dans son utilisation la plus simple, elle ne prend que deux arguments. Pour tracer la courbe définie par un tableau d'abscisses `x` et un tableau d'ordonnées `y`, entrer la commande :

```
--> plot(x, y);
```


Ici `x` et `y` sont des vecteurs : il faut donc impérativement qu'ils aient même dimension.

La fonction `plot` permet de tracer en une seule commande plusieurs couples de variables

```
--> plot(x1, y1, x2,y2);
```

chacun des couples (`xi`, `yi`) devant avoir même dimension.

On peut préciser l'aspect de la courbe (couleurs, type de ligne) en ajoutant, après chaque couple de vecteurs, une chaîne de caractères contenant des options.

```
--> plot(x1, y1,'r-',x2,y2,'b..');
```

la première courbe (`x1`, `y1`) est rouge est continue, la seconde (`x2`, `y2`) est bleue et pointillée (`help plot` pour les options).

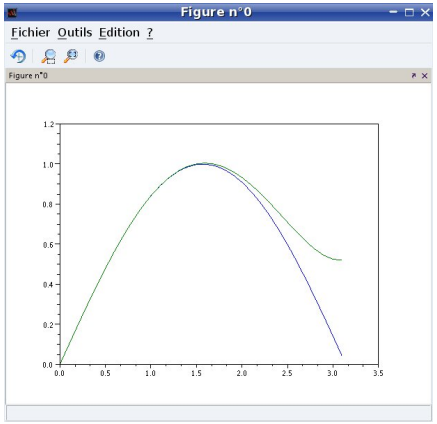
Il existe une autre fonction permettant de faire des tracés 2D s'appelant `plot2d`. Les fonctions `plot` et `plot2d` diffèrent par la syntaxe des options permettant de varier l'aspect des courbes tracées.

```
--> plot2d(x, y);
```

Exemple 4.1 :

Créez le fichier `sinus5.sce` suivant :

```
x = 0:0.1:%pi;  // vecteur des x

y = sin(x);

z = x - x.^3/6 + x.^5/120; //DL

plot(x,y,x,z);
```



Fenêtre graphique créée par le script `sinus5.sce`.

La commande `help plot` vous montre des exemples (idem pour plot2d).

4.2 Ouvrir des fenêtres graphiques distinctes

Le premier appel à la fonction `plot` ouvre une fenêtre graphique identifiée par le numéro zéro (bandeau supérieur de la fenêtre). La commande `scf()` permet de créer une nouvelle fenêtre graphique (vide) avec le numéro 1. La fenêtre nouvellement créée devient la **fenêtre courante** : tout appel à la fonction `plot` sera affiché dans cette fenêtre. Il n'y a qu'une seule fenêtre courante à la fois.

Toute fenêtre graphique est repéré par un numéro affiché dans le bandeau supérieur. Pour sélectionner la figure numéro n comme figure courange il faut utiliser la fonction `scf(n)` (set current figure).

4.2.1 Effacer le contenu d'une fenêtre graphique

Par défaut, les graphes générés par des appels successifs à `plot` sont superposés dans la même fenêtre (la fenêtre courante). Pour effacer le contenu de la fenêtre courante, entrer la commande :

```
--> clf;
```

La superposition automatique des graphes est désactivée par la commande :


```
--> xset( "auto clear", "on" );
```

La superposition peut être réactivée par la même commande, en remplaçant «on» par «off».

Pour effacer le contenu d'une autre fenêtre, disons n, passer son numéro comme argument à clf :

```
--> clf(n);
```

4.3 Modifier la couleur et le style d'un graphe

Pour spécifier la couleur d'une courbe, il faut passer un code couleur (une lettre) comme troisième argument à `plot`. Ainsi, `plot(x,y,'r')` relie les points par une courbe en rouge. Pour afficher les points sous la forme de symboles, il faut passer dans le troisième argument un symbole. (voir `help plot` pour une liste des symboles.)

4.3.1 Donner un titre et légender les axes

La commande `xtitle` permet de donner un titre au graphique de la fenêtre courante. Elle permet également de légender les axes.

```
--> xtitle( "Titre du graphique" ); // pour donner un titre seulement

--> xtitle( "Titre", "Abs", "Ord" ); // pour, en outre, labeler les axes
```

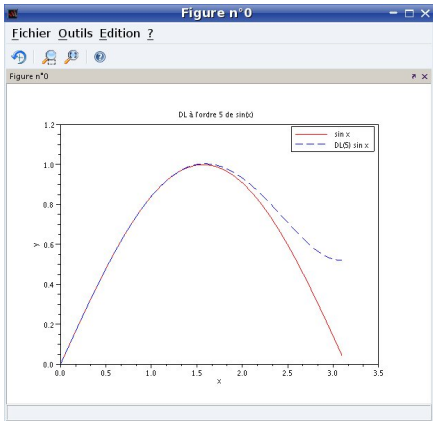
La fonction `legend` permet d'associer de légendes aux courbes superposées sur une même figure.

```
--> legend("legend 1","legend 2");
```

où `"legend 1"`, `"legend 2"` correspondent aux courbes tracées successivement.

Exemple 4.2 :
Notre premier graphique était un peu «nu». Nous allons l'habiller en modifiant quelque peu le fichier `sinus5.sce`.

```
x = 0:0.1:%pi;  // vecteur des x

y = sin(x);

z = x - x.\^3/6 + x.\^5/120; //DL

plot(x,y,'r',x,z,'b--');

legend("sin x","DL(5) sin x");

xlab = "x";

ylab = "y";

titre = "DL à l''ordre 5 de sin(x)"

xtitle(titre,xlab,ylab);
```



Tracé créé par le script `sinus5.sce`

4.4 Imprimer ou enregistrer un graphique

Pour imprimer un graphique, choisir l'item **Print** dans le menu **Fichier** de la fenêtre graphique.

Il est souvent plus commode de sauver le graphique sous la forme d'une image (fichier jpeg, gif ...) ou d'un document (fichier ps ou pdf) afin de la conserver pour une utilisation ultérieure (inclusion dans un compte-rendu ou un rapport par exemple). Pour enregistrer un graphique dans un fichier, choisir l'item **Exporter vers** dans le menu **Fichier** de la fenêtre graphique :
– saisir le nom de fichier souhaité
– choisir le format de sortie (option "filtre")
Le fichier est enregistré dans le répertoire courant par défaut.

Il existe des commandes scilab équivalentes, permettant de sauver un graphique dans une image : `xs2jpg` (jpeg) ou `xs2gif` (gif).

Exemple 4.3 :

```
--> xs2jpg(0,'image0.jpg')  // Sauve la figure 0 come l'image image0.jpg

--> xs2gif(1,'image1.gif')  // Sauve la figure 1 come l'image image1.gif
```

Il est également possible d'enregistrer un graphique dans un format propre à Scilab. Pour cela, choisir l'item **Enregistrer** dans le menu **Fichier** de la fenêtre graphique. Entrer un nom de

fichier (extension .scg). Le graphique peut alors être réaffiché ultérieurement lors de n'importe quelle cession en choisissant l'item **Load** dans le menu **Fichier** puis en sélectionnant le nom du fichier dans la boîte de dialogue qui apparaît. Par contre, on ne peut utiliser les fichiers `.scg` hors de scilab.

Chapitre 5 Fonctions scilab

Il est très utile de pouvoir définir ses propres fonctions. Une fonction réalise une certaine tâche à l'aide d'une commande. Toute fonction est invoquée par une commande :

`[x, y, ... z] = nom_de_la_fonction(a, b, c)`

où `a, b, c` sont les arguments d'entrée (inputs), `x, y, z` les arguments de sorties (output). Certaines fonctions effectuent un calcul numérique : les arguments de sortie sont alors le résultat du calcul compte tenu des arguments d'entrée. Des fonctions peuvent également définir des tâches autres que numériques (graphique, création de fichier, etc).

Exemple 5.1 :

Scilab possède un grand nombre de fonctions prédéfinies.

```
--> y = sin(x);    // une fonction numérique

--> write(data1,x) // une fonction non-numérique
```

Remarque Le nombre d'arguments d'entrée ou de sortie peut éventuellement être nul (voir le dernier exemple).

5.1 Écriture d'une fonction

Il existe deux façons de définir une fonction Scilab :

1. soit dans un script, ou en mode interactif, à l'aide de la commande `deff`,
2. soit en écrivant le code dans un fichier que l'on chargera lors de l'exécution d'un script (avant l'appel à la fonction).

La première méthode est commode pour définir de petites fonctions que l'on n'utilisera qu'au sein du script dans lequel elles sont définies. La seconde méthode permet de réaliser des tâches plus complexes, et d'utiliser les fonctions dans n'importe quel script.

5.1.1 La fonction `deff`

La fonction `deff` est invoquée de la façon suivante :

```
--> deff("[x,y...] = ma_fonction(a,b,...)",commande);
```

Le premier argument de `deff` décrit la syntaxe d'appel de la fonction, le second argument, `commandes`, est une chaîne de caractères décrivant les opérations effectuées.

Exemple 5.2 :

```
--> deff("[z] = plus(a,b)","z = a+b");

--> s = plus(3,18)
 s  =

    21.
```

Remarque importante : Les noms de variables utilisés pour définir les opérations réalisées par la fonction ne sont reconnus que par la fonction. Ils n'ont aucune existence en dehors de la fonction. Ainsi, dans l'exemple précédent, `z` n'existe que dans la fonction, Aucune variable `z` n'existe en mémoire.

5.1.2 Écrire une fonction dans un fichier

La même fonction `plus` peut être définie dans un fichier. Les fonctions sont sauvegardées dans des fichiers d'extension `.sci`.

Toutes les fonctions définies dans un fichier commence par le mot clé `function` et se termine par `endfunction`. Voici la fonction `plus.sci` (attention au «u» de function).

```
function [z] = plus(a,b)

z = a + b;

endfunction
```

Pour être invoquée dans un script, la fonction doit être chargée en mémoire. Pour celà, on utilise la fonction `exec` :

```
exec("plus.sci");
```

La fonction `plus` peut alors être utilisée dans le script.

```
exec("plus.sci")

un_et_un_font = plus(1,1)
```

5.2 Librairie de fonctions

On peut aisément créer des bibliothèques regroupant un ensemble de fonctions. Une façon simple de procéder est de créer un fichier, appelons le `ma_biblio.sci`, contenant plusieurs fonctions :

```
function [x,y] = fonc_1(a,b...)
...
...
\endfunction

function [x,y] = fonc_2(p,q...)
...
...
\endfunction
........
function [x,y] = fonc_k(p,q...)
...
...
\endfunction
```

Toutes les fonctions contenues dans le fichier `ma_biblio.sci` seront chargées, donc utilisables dans un script ou en mode interactif, par la commande :

```
exec("ma_biblio.sci")
```

Chapitre 6 Probabilité et statistique

Scilab possède des fonctions utiles pour le calcul des probabilités et les applications statistiques[1].
Un manuel (en anglais) sur le calcul des probabilités se trouve à l'adresse suivante : http://www.scilab.org/content/download/1105/10844/file/introdiscreteprobas.pdf

6.1 Générateurs de nombre aléatoires

Il existe deux fonctions scilab permettant de générer des matrices de nombres aléatoires :

1. `rand(n,m,'loi')` où `n` est le nombre de ligne de l la matrice, `m` le nombre de colonne, `loi` une chaîne de caractères décrivant la loi de probabilité choisie : `loi` = 'uniform' ($\mathcal{U}(0,1)$) ou 'normal' ($\mathcal{N}(0,1)$). Par défaut (si on ne précise rien) `loi` = 'uniform'.
2. `grand` : fonction beaucoup plus générale que `rand` (`help grand` pour voir les syntaxes).

Exemple 6.1 :

```
--> X = rand(100, 1); // génère un vecteur colonne de 100 nombres
                      // uniformément distribués

--> Y = rand(100, 10,'normal'); // génère une matrice 100x10
                    // constituée de  nombres normalement distribués

--> Z = grand(1,1024,'bin',10,0.3) // génère un vecteur de 1024 nombres
                    // suivant une distribution binomiale (10,0.3)
```

6.2 Calculer une moyenne, une variance, un écart type

La fonction `mean` permet de calculer des moyennes de valeurs stockées dans des vecteurs et des matrices. Pour les matrices, il est possible de procéder au calcul suivant les lignes ou les colonnes :

[1] Les notions décrites dans ce chapitre seront bien sûr abordée en cours

```
--> mx   = mean(x) ; // moyenne des coefficients du vecteur x

--> mm   = mean(m); // moyenne des coefficients de la matrice m

--> mmc  = mean(m,'c'); // moy vecteur des moyennes des lignes

--> mmr  = mean(m,'r'); // moy vecteur des moyennes des colonnes

--> mm1l = mean(m(1,:)); // moyenne de la première ligne

--> mm2c = mean(m(:,2)); // moyenne de la deuxième colonne
```

Il existe des fonctions de syntaxe identique à `mean` pour calculer les variances (fonction `variance` ou les écart types (fonction `stdev`).

```
--> vx  = variance(x) ; // variance des coefficients du vecteur x

--> vm  = variance(m);  // variancedes coefficients de la matrice m

--> vmc = variance(m,'c'); // variances des lignes de m (size(m,2) colonnes)

--> std_m2c = stdev(m(:,2)); // écart type de la deuxième colonne de m

--> std_m2c = sqrt(variance(m(:,2))); // idem
```

6.3 Déterminer l'histogramme d'un ensemble de valeurs

La fonction `histplot` calcule et affiche un histogramme des valeurs d'un vecteur. Le premier argument à passer à la fonction est soit le nombre de classes à utiliser soit un vecteur donnant les bornes de ces classes :

```
--> x = rand(1, 100); // x vecteur de 100 valeurs comprises entre 0 et 1

--> histplot(4, x); // histogramme réalisé avec 4 classes

--> histplot([0:0.2:1], x); // histogramme des classes 0-0.2, 0.2-0.4, etc.
```


Si l'on souhaite faire plus qu'un simple affichage graphique de l'histogramme, il faut utiliser la fonction `dsearch` pour récupérer dans un tableau les effectifs des classes. Cette fonction prend deux arguments : le vecteur des valeurs à classer et le vecteur des bornes des classes. On ne récupère en général que les deux premiers paramètres sur les trois qu'elle retourne, le deuxième étant précisément le tableau des effectifs :

```
--> x = rand( 1, 100 ); // x vecteur de 100 valeurs comprises entre 0 et 1

--> nclasses = 3;

--> bornes = linspace( 0.0, 1.0, nclasses+1 );

--> [idx,histo] = dsearch(x,bornes); // histo: tableau des effectifs
                                     // des classes définies par bornes.
```

6.4 Déterminer les paramètres d'une droite de régression

On dispose de n couples de points dont les abscisses et les ordonnées sont stockées dans deux vecteurs colonnes x et y, (une colonne et n lignes). Les commandes ci-dessous permettent de déterminer et d'afficher les données ainsi que la droite de régression pour ces données :

```
--> [a,b] = reglin(x, y); // calcul des paramètres de la droite

--> plot(x, y,'+');     // affichage des points sous la forme de croix

--> plot(x, a*x+b,'b'); // tracé de la droite en bleu
```

Un certain nombres de fonctions scilab sont disponibles pour évaluer la fonction de répartition des principales distributions (binomiale, Poisson, Student, normale, χ^2, etc.).
Taper `-> help cdf` pour afficher la liste des commandes correspondantes.

6.5 La bibliothèque proba-lib

Il n'y a pas de fonction scilab permettant de calculer ou d'afficher les densités de probabilités de toutes les lois usuelles [2]. La bibliothèque `proba-lib` (une bibliothèque de fonctions pour les

[2] À l'exception de la fonction `binomial`

probabilités) contient les fonctions manquantes. Voir la section **??** pour le chargement de cette bibliothèque dans scilab.

Pour les lois discrètes, les fonctions suivantes retournent le tableau des valeurs $\Pr[X = k]$, k étant un entier ou un vecteur d'entiers :

```
--> pr = pdfpoi(k,lambda) // Loi de Poisson de paramètre m

--> pr = pdfbin(k,p,n)    // Loi binomiale de paramètres n et p
```

Pour les lois continues, les fonctions suivantes retournent les valeurs de la densité de probabilité (ddp) aux positions contenues dans le vecteur `x` :

```
--> y = pdfnor(x,mu,sigma) // Loi normale de moyenne m et variance s2

--> y = pdfstu(x,n)        // Loi de Student à n ddl

--> z = pdfchi(x,n)        // Loi du chi^2 à n degrés de liberté (ddl)
```

On utilisera par exemple les instructions suivantes pour tracer la ddp de la loi du χ^2 à 3 ddl à partir de 100 positions équiréparties sur l'intervalle 0–10 :

```
--> x = linspace(0, 10, 100);

--> p = pdfchi(x,3);

--> plot(x, p);
```

6.5.1 Charger la bibliothèque proba-lib

La bibliothèque `proba-lib` est une collection de fonctions utiles (voir section 5.2). Ces fonctions sont écrites dans le fichier `proba-lib.sci`. La commande `exec` permet de charger en mémoire les fonctions de la bibliothèque.

```
exec("proba-lib.sci",-1)
```

l'argument `-1` rendant le chargement «muet».

Index