INITIATION AU LOGICIEL SCILAB

Christophe Chalons

Initiation au logiciel Scilab

La version de ce document, datée du 16/01/06, est provisoire. Toute remarque, suggestion ou correction est la bienvenue. Merci d'envoyer un email : chalons@math.jussieu.fr

1 Introduction

Scilab (*Scientific Laboratory*) est un logiciel de calcul numérique développé par l'Institut National de Recherche en Informatique et en Automatique (INRIA). Très proche de l'environnement Matlab, il permet, grâce à ses nombreuses fonctions préprogrammées et instructions graphiques, de mettre en pratique et de résoudre rapidement la plupart des problèmes rencontrés dans l'enseignement des mathématiques appliquées (résolution de systèmes linéaires et non linéaires, calcul de valeurs propres et de vecteurs propres, résolution d'équations différentielles, optimisation...). Muni de son propre langage de programmation, simple et agréable, il constitue donc pour l'ingénieur et le mathématicien appliqué une excellente alternative (ou tout au moins un intermédiaire incontournable) à tout langage compilé comme le Fortran ou le C. Des sous-programmes écrits en Fortran ou en C peuvent par ailleurs être facilement liés à Scilab.

Dans l'enseignement, Scilab est souvent préféré au très populaire environnement Matlab. En voici la principale motivation : Scilab est un logiciel libre. Il est téléchargeable gratuitement à partir de l'URL

`http://www.rocq.inria.fr.fr/scilab/`

et est disponible sur tout type standard de machine : PC Windows, Linux, Unix, MacIntosh. Les étudiants ont donc accès au logiciel sur leur ordinateur personnel.

Le présent document ne constitue en aucun cas une présentation complète de Scilab. Il ne décrit que quelques unes des commandes les plus courantes du logiciel, et n'aborde donc qu'une petite partie de ses possibilités. Il devrait toutefois permettre au lecteur de se familiariser avec l'environnement. Pour cela, il est fortement conseillé de lire ce document devant son ordinateur, et de saisir les commandes proposées afin d'en observer les effets.

Pour conclure cette introduction, mentionnons qu'il existe de nombreux documents consacrés à Scilab, disponibles pour la plupart sur le réseau. Nous renvoyons par exemple le lecteur aux références [1], [2], [3], [4], celles-ci ayant largement inspiré l'écriture du présent document.

2 Mes premières commandes Scilab

Après avoir lancé le logiciel, la fenêtre Scilab apparaît à l'écran. La partie supérieure consiste en un menu (dont on discutera l'utilisation au cours des différentes sections) :
`File  Control  Demos  Graphic Window 0  Help`
tandis que la partie inférieure, dédiée à la saisie des instructions, ressemble à :

```
                    ==========
                    scilab-2.7
          Copyright (C) 1989-2003 INRIA/ENPC
                    ==========


Startup execution:
  loading initial environment

-->
```

L'invite `-->` indique que Scilab est prêt à recevoir vos instructions. Chaque instruction, une fois saisie au clavier, est envoyée au logiciel (pour exécution) par un retour-chariot (touche ENTER).
Par exemple la commande :

```
--> 1 + 1  // ma premiere commande Scilab : une somme
```

(suivie d'un retour-chariot) produit le résultat :

```
 ans  =
    2.
```

Ainsi, on remarque dès à présent que :

- le résultat d'un calcul est affecté par défaut à la variable `ans` (pour "answer"). `ans` contient donc le résultat du dernier calcul non affecté à une variable.
- dans une ligne de commande, tout ce qui suit `//` est ignoré, ce qui est fort utile pour insérer commentaires et explications.

De manière analogue, la commande :

```
--> p = 2 * 3  // ma deuxieme commande Scilab : un produit
```

produit le résultat :

```
 p  =
    6.
```

Ici, la variable `p` a été définie et le résultat de l'opération `2*3` lui a été affecté.
Bien entendu, l'ensemble de ces résultats est gardé en mémoire pour un usage ultérieur. Ainsi, la commande

```
--> e = p^ans  // ma troisieme commande Scilab : une puissance
```

produit le résultat attendu :

```
 e  =
    36.
```

Lorsqu'une instruction se termine par un point-virgule, elle est exécutée mais le résultat n'apparaît pas à l'écran. Ainsi,

```
--> a = 100 - 50; // ma quatrieme commande Scilab : une soustraction
--> b = 2; // ma cinqieme commande Scilab : une affectation
--> c = a / b // ma sixieme commande Scilab : une division
```

produit le résultat :

```
 c  =
    25.
```

Pour voir le contenu de la variable `a`, on tape simplement

```
--> a
```

pour obtenir le résultat :

```
 a  =
    50.
```

Plusieurs commandes peuvent être placées sur une même ligne, pourvu qu'elles soient séparées par des points virgules (ou simplement des virgules pour un affichage à l'écran) :

```
--> zero = log(1), un = exp(0); deux = sqrt(4), trois = abs(un-4);
```

produit le résultat :

```
 zero  =
    0.
 deux  =
    2.
```

Inversement, une instruction peut être écrite sur plusieurs lignes pourvu que chaque ligne devant être poursuivie se termine par `..` ou `...` :

```
--> quatre = ...
--> un + trois
```

donne :

```
 quatre  =
```

```
4.
```

Notons ici l'utilisation des fonctions usuelles fournies par Scilab `log`, `exp`, `sqrt` et `abs` (voir également le tableau 1 pour une liste non exhaustive). Notons que les variables `zero` et `un` auraient pu être définies à l'aide des constantes mathématiques prédéfinies (leur valeur ne peut pas être modifiée) π (`%pi`) et e (`%e`) :

```
--> zero = sin(%pi); un = log(%e);
```

Nous terminons cette section en notant que toutes les instructions entrées par l'utilisateur depuis le début d'une session Scilab sont automatiquement gardées en mémoire. Elles sont accessibles par pressions successives de la touche ↑ et peuvent être corrigées et réutilisées sans être retapées entièrement.

3 Utilisation de l'aide en ligne

Scilab dispose d'une aide en ligne. Il est indispensable de savoir et penser à l'utiliser car il est très difficile, sinon impossible, de retenir les options et la syntaxe d'utilisation des nombreuses fonctions préprogrammées du logiciel. D'une manière générale, gardons à l'esprit qu'il est souvent plus simple de savoir où trouver une information que de la retenir, pour un résultat souvent équivalent.

L'accès à l'aide en ligne se fait en cliquant sur le bouton `Help` du menu de la fenêtre Scilab, ou de manière équivalente en tapant la commande `help` (ou encore `help()`) directement dans la fenêtre Scilab. Une nouvelle fenêtre apparaît alors. Sa partie supérieure consiste en un menu dont on ne détaillera pas l'utilisation, tandis que sa partie inférieure regroupe un certain nombre de rubriques aux noms évocateurs (`Graphics Library`, `Linear Algebra`, ...). En cliquant sur l'un de ces sous-menus, la liste de toutes les fonctions préprogrammées correspondantes du logiciel apparaissent. Il suffit alors de cliquer sur le nom d'une fonction pour obtenir le détail de son utilisation (syntaxe, options, ...). Notons qu'il est possible d'obtenir directement le détail d'une fonction dont on connaît déjà le nom (par exemple la fonction `eye`) en saisissant directement dans la fenêtre Scilab la commande `help eye`. La commande `apropos` permet quant à elle d'obtenir toutes les fonctions relatives à un mot clef : saisir par exemple la commande `apropos matrix` (dans la fenêtre Scilab).

La suite de ce polycopié décrit (parfois très brièvement) quelques commandes Scilab. En utilisant l'aide en ligne, vous pourrez en découvrir des utilisations plus avancées.

4 Les types de données

Scilab est un langage faiblement typé. Nous décrivons dans cette section les types scalaires suivants : les nombres réels et complexes, les booléens, les chaînes de caractères et les polynômes. Il existe aussi les rationnels (quotients de deux polynômes) et les listes. Les fonctions sont abordées dans la section 6.

4.1 Les nombres réels et complexes

La manipulation des nombres réels sous Scilab ne présente aucune difficulté. Ils sont traités en double précision, la précision machine étant donnée par la constante prédéfinie `%eps`, de l'ordre de 10^{-16}. Les opérations classiques (addition, soustraction,...) sont disponibles et s'utilisent comme dans la section précédente. Le tableau 1 rassemble quelques fonctions usuelles proposées par Scilab. Bien entendu, cette liste n'est pas exhaustive : on trouve aussi des fonctions liées à la fonction Γ, des fonctions Erreur, des fonctions de Bessel...

Fonction	Description
abs	valeur absolue
exp	exponentielle
log	logarithme népérien
log10	logarithme en base 10
cos	cosinus (argument en radian)
sin	sinus (argument en radian)
tan	tangente (argument en radian)
cotg	cotangente (argument en radian)
acos	cosinus inverse
asin	sinus inverse
atan	tangente inverse
cosh	cosinus hyperbolique
sinh	sinus hyperbolique
tanh	tangente hyperbolique
acosh	cosinus hyperbolique inverse
asinh	sinus hyperbolique inverse
atanh	tangente hyperbolique inverse
sqrt	racine carrée
floor	partie entière ($E(x) \leq x < E(x) + 1$)
ceil	partie entière supérieure ($E(x) - 1 < x \leq E(x)$)
int	partie entière anglaise (floor si $x > 0$, ceil si $x \leq 0$)

Tableau 1 : Quelques fonctions usuelles de Scilab

Scilab permet de manipuler tout aussi facilement les nombres complexes. On utilise pour cela la variable prédéfinie `%i` représentant le nombre imaginaire $i = \sqrt{-1}$. Les opérations classiques restent disponibles. Ainsi, la commande

```
--> d = 1 + 2*%i; e = 1 - 2*%i; f = d * e
```

donne naturellement :

```
 f  =
    5.
```

tandis que l'instruction

```
--> %i^%i - exp(-%pi / 2)
```

conduit à :

```
 ans  =
    0.
```

(rappelons en effet que $i = e^{i\pi/2}$). Le tableau suivant donne les syntaxes Scilab des fonctions mathématiques classiques relatives aux nombres complexes.

Fonction	Description
real	partie réelle
imag	partie imaginaire
abs	module
phasemag	argument (en degré)
conj	conjugué

Tableau 2 : Quelques fonctions relatives aux nombres complexes

Il est important de souligner que les fonctions du tableau 1 s'appliquent également aux nombres complexes. En conséquence, Scilab accepte d'évaluer la racine carrée d'un nombre négatif en renvoyant **l'une des deux** racines complexes de ce nombre (l'autre est simplement l'opposé). La commande

```
--> sqrt(-1)
```
donne alors :
```
 ans  =
    i
```

4.2 Les booléens

Les booléens *true* et *false* sont représentés par les variables prédéfinies `%t` et `%f`. Les opérateurs logiques associés sont donnés dans le tableau suivant

Opérateur	**Description**
&	et
|	ou
~	non
==	égal
<>	différent
≤	plus petit ou égal à
≥	plus grand ou égal à
<	plus petit que
>	plus grand que

Tableau 3 : Opérateurs logiques

En particulier, il est important de ne pas confondre l'opérateur d'affectation `=` utilisé jusqu'à présent, et l'opérateur d'égalité logique `==`. Les instructions suivantes fournissent quelques illustrations.
```
--> 2 == 3  // 2 est-il egal a 3 ?
 ans  =
  F
--> a = (2 < 3)  // 2 est-il plus petit que 3 ?
 a  =
  T
--> b = %t;
--> (~a | ans) & b
 ans  =
  F
```
On remarque en particulier que les booléens *true* et *false* sont représentés à l'affichage par les lettres `T` et `F`.

4.3 Les chaînes de caractères

Les chaînes de caractères sont délimitées par des apostrophes ou des guillemets. Lorsqu'une chaîne contient un tel caractère (apostrophe ou guillemet), celui-ci doit être doublé pour être correctement représenté :

```
--> string1 = " C''est ""cool"""
 string1  =
  C'est "cool"
```
Il est possible de concatener deux chaînes de caractères à l'aide de l'opérateur `+` :
```
--> string1 + ", non ?"
 ans  =
  C'est "cool", non ?
```

La fonction `length` retourne la longueur d'une chaîne de caractères :

```
--> length(string1)
 ans  =
    13.
```

La fonction `part` permet d'extraire un ou plusieurs caractères :

```
--> part(string1,5)
 ans  =
 s
```

La fonction `string` transforme une variable en chaîne de caractères :

```
--> 3  // la variable ans va correspondre au nombre reel 3
 ans  =
    3.
--> string(ans)  // la variable ans va correspondre au caractere 3
 ans  =
 3
```

Le contenu d'une chaîne de caractères peut aussi être interprété comme une expression à évaluer ou une instruction à exécuter. Nous utilisons pour cela les fonctions `evstr` et `execstr`. En voici une illustration :

```
--> x = 2; y = 3; string2 = "x + y"; string3 = "z = x * y";
--> evstr(string2)
 ans  =
    5.
--> execstr(string3)
```

La commande `evstr(string2)` évalue l'expression `x + y` (le résultat est affecté à la variable `ans` et affiché, sauf si l'instruction se termine par un point-virgule), tandis que `execstr(string3)` définie la variable `z` et lui affecte le résultat de l'opération `x * y`. On vérifie alors très simplement le contenu de la variable `z` :

```
--> z
 z  =
    6.
```

On note que la commande `execstr(string2)` n'est pas satisfaisante puisque le résultat de l'opération `x + y` n'est ni affiché à l'écran, ni affecté à la variable `ans` (celle-ci devient même indéfinie). Inversement, la commande `evstr(string3)` n'est pas autorisée.
Le tableau suivant rassemble les fonctions évoquées dans cette section.

Fonction	**Description**
+	concaténer
length	longueur
part	extraire
string	transformer en chaîne
evstr	évaluer une expression
execstr	exécuter une instruction

Tableau 4 : Quelques fonctions relatives aux chaînes de caractères

4.4 Les polynômes

Scilab propose par défaut un polynôme élémentaire prédéfini : `%s`. A partir de ce polynôme élémentaire, on utilise les opérations classiques pour définir un polynôme quelconque :

```
--> p1 = 1 + 2 * %s - 3 * %s^2
 p1  =
                2
    1 + 2s - 3s
```

```
--> p2 = (%s - 1) * (%s - 2) * (%s - 3)
 p2  =
                2    3
  - 6 + 11s - 6s  + s
--> p1+p2
 ans  =
                2    3
  - 5 + 13s - 9s  + s
```

La fonction `coeff` renvoie les coefficients d'un polynôme sous la forme d'un vecteur ligne :

```
--> coeff(p2)
 ans  =
! - 6.    11.    - 6.    1. !
```

La fonction `roots` renvoie les racines d'un polynôme sous la forme d'un vecteur colonne :

```
--> roots(p2)
 ans  =
!   1. !
!   2. !
!   3. !
```

La fonction `poly` permet de construire automatiquement un polynôme d'une variable quelconque à partir d'un vecteur contenant ses coefficients (option `coeff` ou `c`) ou ses racines (option par défaut ou `roots` ou `r`), comme l'illustrent les commandes suivantes (nous renvoyons le lecteur à la section 5 pour plus de détails sur la manipulation des vecteurs dans Scilab) :

```
--> v = [0, 1, 2]  // definition du vecteur ligne v=(0,1,2)
 v  =
!   0.    1.    2 !

--> p3 = poly(v, "x", "coeff")
 p3  =
          2
    x + 2x

--> p4 = poly(v, "t", "roots")
 p3  =
           2   3
    2t - 3t + t
```

Notons qu'il n'est pas possible d'effectuer des opérations sur des polynômes dont la variable est différente : par exemple, les polynômes `p1` et `p3` ne peuvent pas être additionnés.

5 Les vecteurs et les matrices

Les vecteurs et les matrices de nombres réels ou complexes constituent un des types de base de Scilab. Ils interviennent de manière récurrente dans son utilisation, que ce soit dans le cadre de "simples" calculs, d'une programmation plus poussée ou de tracés graphiques. Nous verrons en particulier qu'un des points forts de l'environnement Scilab est la très grande simplicité de la syntaxe liée au calcul matriciel.

5.1 Les vecteurs

Nous décrivons dans ce paragraphe quelques commandes Scilab spécifiques aux vecteurs. Il est important de noter que tout vecteur Scilab (ligne ou colonne) peut être vu comme une matrice particulière (à une ligne ou une colonne), de sorte que bon nombre des commandes

décrites dans le paragraphe suivant dédié aux matrices, notamment celles sur les opérations élémentaires, s'appliquent naturellement aux vecteurs construits ici.

Il existe trois syntaxes principales pour créer un vecteur ligne. Dans la première syntaxe, la liste des éléments, séparés par des espaces ou des virgules, est entourée de crochets. Par exemple, la commande

```
-->v1 = [1, %i, 1 + %i]
```

produit le résultat

```
 v1  =
!   1.  i    1. + i  !
```

De même,

```
-->w1 = [1 2 3]
```

conduit à

```
 w1  =
!   1.   2.   3.  !
```

Afin d'éviter les erreurs, on préfèrera séparer les éléments par des virgules. L'exemple suivant devrait suffir à convaincre le lecteur réticent :

```
-->w2 = [1 -3 2 3]
 w2 =
!   1.   -3.   2.   3  !
-->w3 = [1 - 3 2 3]
 w3  =
!   -2.   2.   3.  !
```

La deuxième syntaxe permet de créer un vecteur ayant un incrément (positif ou négatif) constant entre chacun de ses éléments, le premier élément étant précisé par l'utilisateur, tandis que le dernier est automatiquement déterminé par une valeur limite à ne pas dépasser. La syntaxe est

`x = [premier:increment:limite]` ou `x = premier:increment:limite`.

Lorsque l'incrément est égal à `1`, on peut utiliser simplement

`x = [premier:limite]` ou `x = premier:limite`.

A titre d'illustration :

```
-->x1 = [0:0.4:2]
 x1 =
!   0.   0.4   0.8   1.2   1.6   2.  !
-->x2 = [0:0.3:2]
 x2 =
!   0.   0.3   0.6   0.9   1.2   1.5   1.8  !
-->x3 = 0:5
 x3 =
!   0.   1.   2.   3.   4.   5.  !
```

Notons que la valeur limite ne coïncide pas nécessairement avec le dernier élément du vecteur créé. Si l'on souhaite imposer le premier et le dernier élément du vecteur, il est préférable d'utiliser la troisième syntaxe. Elle permet de même que précédemment de créer un vecteur d'éléments régulièrement espacés, mais dont la taille, le premier élément et le dernier élément sont précisés par l'utilisateur. La syntaxe précise est

`x = linspace(premier,dernier,nombre)` :

```
-->x4 = linspace(0,5,6)
 x4  =
!   0.   1.   2.   3.   4.   5.  !
-->x5 = linspace(0,6,5)
 x5  =
!   0.   1.5   3.   4.5   6.  !
```

Il est possible d'obtenir un vecteur colonne en utilisant la première syntaxe et en séparant chaque élément par un point-virgule :

```
-->v2 = [1; %i; 1 + %i]
 v2  =
!   1.       !
!   i        !
!   1. + i   !
```

ou en utilisant les symboles ' (transposée conjuguée) et .' (transposée non conjuguée) :

```
-->v3 = v1'
 v3  =
!   1.       !
! - i        !
!   1. - i   !
-->v1.'
 ans  =
!   1.       !
!   i        !
!   1. + i   !
```

L'accès à un élément d'un vecteur (ligne ou colonne) se fait simplement en utilisant des parenthèses :

```
-->v1(2)
 ans  =
    i
-->v3(3)
 ans  =
    1. - i
```

L'accès au dernier élément d'un vecteur (ligne ou colonne) se fait à l'aide du symbole $:

```
-->v1($)
 ans  =
    1. + i
```

Enfin, la commande `vec1(vec2)` (respectivement `vec1(vec2) = []`) extrait (respectivement supprime) du vecteur `vec1` les coordonnées dont l'indice est un élément du vecteur `vec2` :

```
-->y1 = [0:1:10]; y2 = [1:3:10];
-->y1(y2)
 ans  =
!   0.   3.   6.   9.  !
-->y1(y2) = []
 y1  =
!   1.   2.   4.   5.   7.   8.   10.  !
```

5.2 Les matrices

Dans ce paragraphe, nous généralisons la notion de vecteurs en décrivant quelques commandes liées aux matrices.

5.2.1 Construction, extraction, suppression

La construction d'une matrice suit une syntaxe analogue à celle des vecteurs : les composantes d'une même ligne sont séparées par des virgules (ou des espaces), chaque ligne (sauf la dernière) se terminant par un point virgule.

```
-->A = [1,2,3;4,5,6;7,8,9]  // definition d'une matrice 3 x 3
 A  =
!   1.    2.    3. !
!   4.    5.    6. !
!   7.    8.    9. !
```

Il est possible de séparer les lignes d'une matrice par des retours-chariots :

```
-->B = [1,2,3,4;
-->5,6,7,8]     // definition d'une matrice 2 x 4
 B  =

!   1.    2.    3.    4. !
!   5.    6.    7.    8. !
```

Pour extraire un élément d'une matrice, il suffit de mentionner ses indices de ligne et de colonne entre parenthèses :

```
-->A23 = A(2,3)
 A23  =
    6.
```

Il est aussi possible d'extraire entièrement une ligne ou une colonne à l'aide du symbole `:`, par exemple la deuxième ligne :

```
-->L2 = B(2,:)
 L2  =
!   5.    6.    7.    8. !
```

ou la dernière colonne :

```
-->C4 = B(:,$)
 C4  =
!   4. !
!   8. !
```

Plus généralement, il est possible d'extraire ou de supprimer plusieurs éléments d'une matrice. Ainsi, si `vl` et `vc` désignent deux vecteurs d'entiers, la commande `M(vl,vc)` (respectivement `M(vl,vc) = []`) extrait (respectivement supprime) de la matrice `M` les éléments dont l'indice de ligne est dans `vl` et dont l'indice de colonne est dans `vc`. Pour extraire (respectivement supprimer) les lignes d'indices dans `vl` ou les colonnes d'indices dans `vc`, on utilise la commande `M(vl,:)` ou la commande `M(:,vc)` (respectivement `M(vl,:) = []` ou `M(:,vc) = []`).

La syntaxe de construction d'une matrice proposée au début de ce paragraphe peut être vue comme celle associée à un assemblage (empilement et/ou concaténation) de nombres réels ou complexes, c'est-à-dire à un assemblage de matrices de taille 1×1. En réalité, Scilab autorise l'assemblage de matrices de taille quelconque, pourvu que les dimensions coïncident.

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

En particulier, si `v` et `w` sont deux vecteurs lignes, la commande `[v,w]` les concatènera tandis que la commande `[v;w]` les empilera. Rappelons en effet que les vecteurs ne sont rien d'autre que des matrices particulières.

5.2.2 Matrices particulières

Des fonctions prédéfinies permettent de construire des matrices particulières. Par exemple, la commande `eye(n,p)` permet d'obtenir une matrice de taille $n \times p$ avec des 1 sur la diagonale principale (éléments (i, j) tels que $i = j$) et des 0 partout ailleurs. Lorsque $n = p$, on obtient la matrice identité de taille n.

```
-->Id23 = eye(2,3)
 Id23  =
!   1.    0.    0. !
!   0.    1.    0. !
```

Pour obtenir un matrice de taille $n \times p$ ne contenant que des 0 (respectivement que des 1), on utilise la commande `zeros(n,p)` (respectivement `ones(n,p)`). La commande `rand(n,p)` permet quant à elle d'obtenir une matrice de taille $n \times p$ à coefficients aléatoires uniformes sur $[0, 1[$.
Notons que les dimensions n et p de ces matrices particulières peuvent être données par l'intermédiaire d'une matrice (déjà définie dans la session) que l'on passe en argument. Par exemple :

```
-->ones(ans)
 ans  =
!   1.    1.    1.    1. !
!   1.    1.    1.    1. !
!   1.    1.    1.    1. !
```

La commande `toeplitz(vc,vl)` retourne la matrice de Toeplitz (*i.e.* à diagonales constantes) dont la première ligne est le vecteur `vl` et dont la première colonne est le vecteur `vc` (`vl(1)` et `vc(1)` doivent coïncider). La commande `toeplitz(v)` retourne la matrice de Toeplitz symétrique dont la première ligne est le vecteur `v`.

```
-->v1 = [1,2,3,4]; v2 = [1;10;100];
-->toeplitz(v1,v2)
 ans  =
!   1.    10.   100. !
!   2.    1.    10.  !
!   3.    2.    1.   !
!   4.    3.    2.   !
-->toeplitz(v2,v1)
 ans  =
!   1.     2.    3.    4. !
!   10.    1.    2.    3. !
!   100.   10.   1.    2. !
```

La commande `diag(v)` permet d'obtenir une matrice diagonale dont les éléments diagonaux sont ceux du vecteur (ligne ou colonne) `v` :

```
-->diag(v1)
 ans  =
!   1.    0.    0.    0. !
!   0.    2.    0.    0. !
!   0.    0.    3.    0. !
!   0.    0.    0.    4. !
```

Appliquée à une matrice, la fonction `diag` permet d'en extraire la diagonale principale sous la forme d'un vecteur colonne :

```
-->diag(ans)
 ans  =
!   1. !
!   2. !
!   3. !
!   4. !
```

Enfin, la commande `triu(A)` (respectivement `tril(A)`) retourne la matrice triangulaire supérieure (respectivement inférieure) associée à la matrice `A`.

5.2.3 Fonctions usuelles

Scilab dispose de plusieurs fonctions relatives aux notions de base d'algèbre linéaire. Quelques unes d'entre elles sont regroupées dans le tableau suivant :

Fonction	Description
A'	transposée conjuguée de la matrice A
A.'	transposée de la matrice A
det(A)	déterminant de la matrice A
trace(A)	trace de la matrice A
inv(A)	inverse de la matrice A
rank(A)	rang de la matrice A
cond(A)	conditionnement de la matrice A
norm(A)	norme de la matrice A
spec(A)	spectre de la matrice A
kernel(A)	noyau de la matrice A
svd(A)	valeurs singulières de la matrice A
poly(A,"x")	polynôme caractéristique de la matrice A
expm(A)	exponentielle de la matrice A

Tableau 5 : Quelques fonctions usuelles d'algèbre linéaire

Concernant la fonction `norm`, il est possible de choisir le type de la norme à l'aide d'un second argument optionnel : `norm(A,2)` calcul la norme 2 de la matrice `A`, `norm(A,1)` la norme 1, `norm(A,'inf')` la norme infinie et `norm(A,'fro')` la norme de Frobenius. Par défaut, `norm(A)` calcule la norme 2 de la matrice `A`. Lorsque la fonction `norm` est appliquée à un vecteur, `norm(v,p)` calcul la norme l^p du vecteur `v` tandis que `norm(v,'inf')` calcul la norme infinie du vecteur `v`. Par défaut, `norm(v)` calcule la norme l^2 du vecteur `v`.
La commande `spec(A)` renvoie par défaut un vecteur colonne contenant les valeurs propres de la matrice `A` comme l'illustre l'exemple suivant :

```
-->A = [1,2;3,4];
-->spec(A)
 ans  =
!   5.3722813 !
! - 0.3722813 !
```

Il est également possible d'obtenir les vecteurs propres associés à l'aide de la commande `[MVP,Mvp] = spec(A)`. La matrice `Mvp` est alors une matrice diagonale formée des valeurs propres de `A`, tandis que `MVP` est une matrice inversible contenant les vecteurs propres associés :

```
-->[MVP,Mvp] = spec(A)
 Mvp  =
!   5.3722813    0.        !
!   0.         - 0.3722813 !
 MVP  =
!   0.4159736  - 0.8245648 !
!   0.9093767    0.5657675 !
```

Scilab dispose également de plusieurs fonctions vectorielles qui s'appliquent à une matrice ou à un vecteur et qui retournent un scalaire.

Fonction	Description
max(A)	maximum des éléments de A
min(A)	minimum des éléments de A
sort(A)	tri par ordre décroissant des éléments de A
sortup(A)	tri par ordre croissant des éléments de A
sum(A)	somme des éléments de A
prod(A)	produit des éléments de A

Tableau 6 : Quelques fonctions vectorielles

Il est possible d'appliquer une telle fonction colonne par colonne (respectivement ligne par ligne) à l'aide de l'argument optionnel `'r'` (respectivement `'c'`). La fonction retournera alors

un vecteur ligne (respectivement colonne). Par exemple, la commande `prod(A,'r')` retourne un vecteur ligne formé des produits des coefficients dans chaque colonne de `A`.
La fonction `size` produit un vecteur ligne contenant les deux dimensions (nombre de lignes puis nombre de colonnes) de la matrice argument. Le tableau suivant décrit les options de la fonction `size`.

Fonction	**Description**
size(A)	nombre de lignes et de colonnes de A
size(A,'r')	nombre de lignes de A
size(A,'c')	nombre de colonnes de A
size(A,'*')	nombre total d'éléments de A

Tableau 7 : La fonction `size`

La commande `length(A)` fournit le nombre d'éléments d'une matrice réelle ou complexe (la commande `size(A,'*')` donne le même résultat).

Enfin, il est important de souligner que les fonctions numériques proposées dans le tableau 1 s'appliquent aussi à des matrices, composante par composante, comme l'illustre l'exemple suivant :

```
-->A = [0,%pi;%pi,0];
-->cos(A)
 ans  =
!   1.  - 1. !
! - 1.    1. !
```

A ce sujet, le lecteur attentif notera la différence entre la commande `expm(A)` (proposée dans le tableau 5) et la commande `exp(A)`.

5.2.4 Opérations sur les matrices

Une des particularités de Scilab est de disposer d'une syntaxe particulièrement simple pour les opérations matricielles. Pourvu que les dimensions soient compatibles, on additionne, on soustraie, on multiplie, on élève à la puissance des matrices en utilisant simplement les opérateurs classiques `+`, `-`, `*`, `^`. En voici une illustration :

```
-->A = toeplitz([0,1]); B = diag([1,2]); C = ones(2,2);
-->(A+B)*C^2
 ans  =
!   4.    4. !
!   6.    6. !
```

Les opérations s'effectuent suivant les ordres de priorité classiques. Des messages d'erreur sont affichés si une opération impossible est tentée (dimensions incompatibles). Notons toutefois qu'il est possible d'additionner, de soustraire ou de multiplier une matrice de taille quelconque avec un scalaire. Dans ce cas, l'opération s'appliquera terme à terme :

```
-->1 + A
 ans  =
!   1.    2. !
!   2.    1. !
-->2 * A
 ans  =
!   0.    2. !
!   2.    0. !
```

Multiplier terme à terme deux matrices et élever à la puissance une matrice terme à terme sont également possibles. Pour cela, les opérateurs `*` et `^` doivent être précédés par un point, c'est-à-dire remplacés par un `.*` et `.^` :

```
-->A * A
```

```
 ans  =
!   1.    0. !
!   0.    1. !
-->A .* A
 ans  =
!   0.    1. !
!   1.    0. !
```

La division matricielle existe aussi sous Scilab. Tout d'abord, il est possible de diviser une matrice par un scalaire :

```
-->6 * ones(2,2)
 ans  =
!   6.    6. !
!   6.    6. !
-->ans / 12
 ans  =
!   0.5    0.5 !
!   0.5    0.5 !
```

Scilab permet également de résoudre des systèmes linéaires. Ainsi, la commande `A\B` (respectivement `A/B`) fournit une solution du système `AX=B` (respectivement `XB=A`) lorsque les dimensions sont compatibles.
Si `v` est un vecteur ligne (respectivement colonne), `1/v` renvoie un vecteur colonne (respectivement ligne) noté `w` et tel que `v*w=1` (respectivement `w*v=1`). A ce sujet, l'instruction `1./v` est comprise comme `(1.)/v`, ou encore `1/v`. En revanche, l'instruction `(1)./v` divise terme à terme les coefficients de `v`. Elle est donc équivalente à l'instruction `ones(v)./v`, ou encore à l'instruction `v.^(-1)`.
Enfin, si `s` est un réel et `M` une matrice, alors `s/M` fournit la matrice solution de `MX=s*eye(M)` (rappel : `eye(M)` est la matrice de même taille que `M` avec des 1 sur la diagonale et des 0 partout ailleurs).

6 Programmation

Scilab dispose d'un langage de programmation simple et agréable. Il est proche dans sa syntaxe des langages courants comme le Fortran ou le C. La différence essentielle avec ces langages compilés réside dans la non nécessité de déclarer les variables utilisées : on dit que le langage de programmation de Scilab est interprété.

6.1 Les scripts

Lorsque des calculs à effectuer nécessitent un nombre important de commandes, l'utilisateur a tout intérêt à écrire un **script**, c'est-à-dire à saisir ces commandes dans un fichier externe au logiciel Scilab, qu'il exécutera ensuite. L'ensemble sera alors plus lisible et surtout, n'aura pas à être resaisi pour une utilisation ultérieure. Par convention, les noms des fichiers de commandes Scilab sont suffixés par la terminaison `.sce`.
Supposons que les commandes Scilab suivantes aient été saisies dans un fichier nommé `script1.sce` :

```
A = ones(3,3);
b = 6 * ones(3,1);
x = A\b
```

Si le fichier `script1.sce` se trouve dans le repertoire courant (répertoire à partir duquel a été lancé Scilab), il suffit alors de saisir la commande

```
-->;exec"script1.sce";
```

dans la fenêtre Scilab pour que s'affiche la valeur du vecteur `x`, solution du système linéaire `Ax=b` :

```
 x  =
!   1. !
!   2. !
!   3. !
```

Sinon, le nom du fichier doit être remplacé par son chemin d'accès complet.
Il existe une autre méthode pour exécuter un script : dans le menu (partie supérieure de la fenêtre Scilab), sélectionner l'item `File` puis `File Operations`, sélectionner ensuite le fichier désiré (ici `script1.sce`) en changeant de répertoire si nécessaire, puis cliquer sur le bouton `exec`.

6.2 Les fonctions

Scilab dispose d'un nombre important de fonctions préprogrammées, certaines ayant été décrites dans les sections précédentes. Il est possible d'étendre le langage en définissant ses propres fonctions. Comme dans tout langage de programmation, celles-ci permettent à l'utilisateur de regrouper un ensemble d'instructions sous un même nom. Nous décrivons dans cette section les deux approches principales permettant de définir ses propres fonctions.

La première approche consiste à donner la définition d'une nouvelle fonction dans un fichier de commandes (script) ou directement dans la fenêtre Scilab. Pour cela, deux syntaxes sont disponibles.
La première syntaxe utilise la fonction `deff`. Elle est agréable pour une fonction ne comportant qu'un petit nombre d'instructions. Voici un premier exemple :

```
-->deff('[x,y] = myfunction1(a,b)',['x = a + b'; 'y = a - b'])
-->[s1,d1]=myfunction1(3,2)
 d1  =
     1.
 s1  =
     5.
```

On observe que le premier argument de `deff` concerne le prototype de la nouvelle fonction : arguments de sortie entre crochets (`x` et `y`), nom de la fonction (`myfunction1`) et arguments d'entrée entre parenthèses (`a` et `b`). Le second argument est une suite d'instructions formant le corps de la fonction et présentée sous la forme d'un vecteur de chaînes de caractères.
Lorsque le nombre d'instructions formant le corps de la nouvelle fonction à définir devient important, il est possible d'améliorer la lisibilté du script en utilisant la syntaxe suivante :

```
-->function [x,y] = myfunction1(a,b)
-->x = a + b
-->y = a - b
-->endfunction
-->[s2,d2]=myfunction1(5,3)
 d2  =
     2.
 s2  =
     8.
```

La deuxième approche consiste à définir la nouvelle fonction dans un fichier externe au logiciel Scilab, que l'on importera ensuite dans l'environnement. De même que pour les scripts, l'ensemble pourra ainsi être conservé d'une session Scilab à l'autre sans être resaisi. Par convention, les noms des fichiers de fonctions Scilab sont suffixés par la terminaison `.sci`. Ces fichiers peuvent contenir la définition d'une ou plusieurs fonctions.
Supposons que les commandes Scilab suivantes aient été saisies dans un fichier nommé `myfunctions.sci` (la dernière instruction du fichier doit obligatoirement être suivie d'un retour chariot pour être prise en compte) :

```
function [x,y] = myfunction1(a,b)
x = a + b
y = a - b
function [x] = myfunction2(a,b)
x = a * b
```

Si le fichier `myfunctions.sci` se trouve dans le repertoire courant, il faut saisir la commande

```
-->getf"myfunctions.sci"
```

dans la fenêtre Scilab (ou dans un script le cas échéant) pour charger et compiler les nouvelles fonctions. Sinon, le nom du fichier doit être remplacé par son chemin d'accès complet.
A partir de cet instant, les fonctions `myfunction1` et `myfunction2` font partie de l'environnement Scilab (pour la session courante uniquement) et peuvent être utilisées comme toute autre fonction (prédéfinie ou non). Dans la fenêtre Scilab, les commandes :

```
-->[s,d] = myfunction1(0.5,0.5), [p] = myfunction2(0.5,4),
```

produisent alors le résultat attendu :

```
 d  =
    0.
 s  =
    1.
 p  =
    2.
```

Remarque 1
Une fonction peut être passée en argument d'une autre fonction.

Remarque 2
A l'intérieur d'une fonction, le résultat d'une commande n'est pas affiché à l'écran même si celle-ci ne termine pas par un point-virgule. Pour afficher à l'écran le résultat d'une commande effectuée à l'intérieur d'une fonction, on peut utiliser la fonction `disp` (voir l'aide en ligne pour plus de détails). En voici un exemple d'utilisation.

```
-->function [x,y] = myfunction1(a,b)
-->disp('Je vais calculer x et y')
-->x = a + b
-->y = a - b
-->disp(y,'et y = ',x,'Le resultat est x = ')
-->endfunction
-->[s,d]=myfunction1(5,3);
 Je vais calculer x et y
 Le resultat est x =
    8.
 et y =
    2.
```

Attention à l'ordre d'affichage de la fonction `disp`. La commande `printf` (qui est une émulation de la même commande en C) est également disponible.

6.3 Les boucles

6.3.1 La boucle for

La syntaxe d'une boucle `for` est la suivante :

```
for v = M, instruction1, ..., instructionN, end
```

ou `M` est une matrice et `v` un vecteur. La boucle itère sur les colonnes de la matrice `M` (le nombre d'itérations est donc donné par le nombre de colonnes de `M`). A l'itération i, le vecteur v coïncide donc avec la $i$ème colonne de `M`. A chaque itération, les instructions `instruction1`, ...,

`instructionN` sont exécutées. Les instructions peuvent être séparées par des points virgules pour limiter les affichages à l'écran. Notons que `v` est un scalaire dans le cas particulier où `M` est une matrice ligne (*i.e.* un vecteur ligne).
Il est possible (et fortement conseillé dans la plupart des cas) d'améliorer la lisibilité de la boucle en utilisant la syntaxe suivante :

```
for v = M
 instruction1
      ...
 instructionN
end
```

Ici, le passage à la ligne se substitue à la virgule et les instructions peuvent être suivies d'un point virgule pour limiter les affichages. Cette syntaxe peut s'utiliser dans un script ou directement dans la fenêtre Scilab.
Voici deux exemples simples et courants d'utilisation :

```
-->x=0, for k=2:2:6, x = x + k, end
 x  =
    0.
 x  =
    2.
 x  =
    6.
 x  =
    12.
-->x = 1; v=[1,2,3];
-->for k=v
--> x = x * k
-->end
 x  =
    1.
 x  =
    2.
 x  =
    6.
```

6.3.2 La boucle while

La boucle `while` permet de répéter une suite d'instructions tant qu'une condition est vraie, et d'effectuer (une seule fois) une autre suite d'instructions si la condition est fausse. La syntaxe d'une boucle `while` est la suivante :

```
while condition then, instructions1, else, instructions2, end
```

ou `instructions1` et `instructions2` sont des suites d'instructions séparées par des virgules (ou des points virgules) et `condition` est un booléen. Il existe deux variantes dans lesquelles le `then` est ommis ou remplacé par un `do`. Les instructions liées au `else` sont par ailleurs facultatives. La syntaxe

```
while condition, instructions1, end
```

est donc correcte.
De même que précédemment, il est conseillé d'améliorer la lisibilité de la boucle en utilisant (dans un script ou directement dans la fenêtre Scilab) la syntaxe suivante :

```
while condition then
 instructions1
else
 instructions2
end
```

Là encore, le `then` peut être ommis ou remplacé par un `do`.

```
-->x = 0;
-->while (x < 5) then
-->x = x + 1
-->else
-->x = x + 2
-->end
 x  =
    1.
 x  =
    2.
 x  =
    3.
 x  =
    4.
 x  =
    5.
 x  =
    7.
```

6.4 Les tests

6.4.1 Le if-then-else

L'instruction conditionnelle `if` permet d'effectuer une suite d'instructions si une condition est vraie, et d'effectuer une autre suite d'instructions si la condition est fausse. La syntaxe la plus simple d'une instruction `if` est la suivante :

```
if condition then, instructions1, else, instructions2, end
```

ou `instructions1` et `instructions2` sont des suites d'instructions séparées par des virgules (ou des points virgules) et `condition` est un booléen. Deux variantes consistent à ommettre le `then` ou à le remplacer par un `do`. Les instructions liées au `else` sont facultatives. Pour améliorer la lisibilité de l'instruction (dans un script ou directement dans la fenêtre Scilab) la syntaxe suivante est disponible :

```
if condition then
 instructions1
else
 instructions2
end
```

Là encore, le `then` peut être ommis ou remplacé par un `do`.

```
-->x = 0;
-->if (x < 5) then, x = x + 1, else, x = x + 2, end
 x  =
    1.
```

6.4.2 Le select-case

L'instruction conditionnelle `select` permet d'effectuer une suite d'instructions dépendant de la valeur d'une variable. La syntaxe est la suivante :

```
select variable
 case variable1 then
  instructions1
 case variable2 then
  instructions2
      ...
```

```
      ...
 else
   instructionsN
end
```

ou `instructions1`, `instructions2`, ..., `instructionsN` sont des suites d'instructions. Deux variantes consistent à ommettre le `then` ou à le remplacer par un `do`. Les instructions liées au `else` sont facultatives. Scilab teste donc successivement l'égalité de la variable `variable` avec les différentes propositions : ici `variable1`, `variable2`, ... Dès que l'égalité est vraie, les instructions correspondantes sont effectuées et Scilab sort de la construction. Si la variable `variable` ne coïncide avec aucune des propositions, les instructions relatives au `else` sont effectuées.

```
-->x = 0;
-->select x
-->case 1 then
-->x = x + 1;
-->case 2 then
-->x = x + 2;
-->else
-->x = x + 3;
-->end
-->x
 x  =
    3.
```

7 Représentations graphiques

Scilab dispose de multiples fonctions graphiques qu'il n'est pas possible de décrire dans le détail. Dans cette section, nous portons notre attention sur la fonction `plot2d` qui est la procédure de base pour représenter une courbe dans un plan.

7.1 Les fenêtres graphiques

Lorsqu'une instruction graphique comme `plot2d` est lancée, Scilab affiche le dessin dans une fenêtre graphique numérotée qui devient la fenêtre courante, c'est-à-dire celle qui recevra, sauf mention contraire, les futurs instructions graphiques ou règlages de paramètres. Si aucune fenêtre graphique n'est activée, Scilab choisit comme fenêtre courante la fenêtre de numéro 0. La partie supérieure d'une fenêtre graphique consiste en un menu : `File  Zoom  UnZoom  3D Rot.` La partie inférieure est dédiée à l'affichage des courbes.
Il est possible de sélectionner la fenêtre courante (en la créant préalablement si nécessaire) à l'aide de la commande `xset("window",num)` (la fenêtre courante devient la fenêtre de numéro `num` ; si celle-ci n'existait pas, elle est crée par Scilab). L'instruction `xbasc([num1,...,numN])` (respectivement `xdel([num1,...,numN])`) nettoie (respectivement détruit) les fenêtres `num1`, ..., `numN`. Si `[num1,...,numN]` est ommis, Scilab nettoie (respectivement détruit) la fenêtre courante. La fonction `xbasc` est très utile dans la pratique, dans la mesure ou Scilab superpose par défaut les graphiques successifs.
D'une manière générale, Scilab permet de régler tous les paramètres (épaisseur des traits, couleur, ...) de la fenêtre graphique courante à l'aide de la fonction `xset`. Pour plus de détails sur cette fonction, nous renvoyons le lecteur à l'aide en ligne (taper la commande `help xset` dans la fenêtre Scilab). Notons que la plupart des paramètres graphiques peuvent être choisis interactivement via un menu graphique que l'on peut voir apparaître en saisissant la commande `xset()` dans la fenêtre Scilab.

7.2 La fonction `plot2d`

La fonction `plot2d` permet de représenter plusieurs courbes dans le plan. Elle autorise un style propre à chaque courbe. La syntaxe générale est la suivante :
`plot2d(abscisses, ordonnees, [style, cadre, legendes, bornes, graduations])`
Les arguments entre crochets sont facultatifs, mais si l'un d'entre eux est précisé, les précédents devront l'être aussi. La suite de ce paragraphe est consacrée à une description succinte des arguments de la fonction `plot2d`.

• `abscisses` et `ordonnees` sont des matrices de même dimension contenant respectivement les abcisses et les ordonnées des points à représenter. Si on note `npt` (respectivement `nc`) le nombre de lignes (respectivement de colonnes) de ces matrices, `nc` représente le nombre de courbes devant être tracées et `npt` le nombre de points pour toutes ces courbes. Ainsi, les abscisses (respectivement les ordonnées) des points de la $i$ème courbe à représenter correspondent à la $i$ème colonne du vecteur `abscisses` (respectivement `ordonnees`)

• `style` est un vecteur ligne de dimension le nombre de courbes à tracer (ici `nc`). `style(j)` définit la manière de représenter la courbe numéro `j` : si `style(j)`$\leq$ `0` on utilise des marqueurs de code `-style(j)` ; si `style(j)`>`0` on utilise des lignes pleines ou pointillées de couleur `style(j)` (la correspondance des codes de couleurs et des marqueurs figure dans le menu graphique obtenu par la commande `xset()`).

• `cadre` est une chaine de trois caractères `"xyz"`.
Si `x` vaut 1, la légende fournie dans l'argument `legende` sous la forme `"legende1@legende2@...."` est affichée. Si `x` vaut 0, aucune légende n'est affichée.
Le caractère `y` concerne le cadrage : si `y=0`, les bornes courantes sont utilisées ; si `y=1` l'argument `bornes = [xmin,ymin,xmax,ymax]` est utilisé pour redéfinir les bornes ; si `y=2` les bornes sont calculées à partir des valeurs extrêmes des matrices `abscisses` et `ordonnees`.
Le caractère `z` gère l'entourage du graphique : `z=1` permet d'avoir des axes gradués et `z=2` un cadre autour du graphique.

• `legendes` est une chaine de caractères contenant les différentes légendes séparées par `@` : `"legende1@legende2@...."`

• `bornes` est le rectangle de représentation. Il est décrit par les deux coordonnées des coins inférieur gauche `(xmin,ymin)` et supérieur droit `(xmax,ymax)` :
`bornes = [xmin,ymin,xmax,ymax]`.

• `graduations` est un vecteur ligne de 4 entiers permettant de préciser la fréquence des graduations et sous-graduations en abscisse et en ordonnée : avec `[2, 10, 4, 5]`, l'intervalle des abscisses sera divisé en 10, chacun de ces 10 sous-intervalles étant divisé en 2, tandis que pour l'intervalle des ordonnées, il y aura 5 sous-intervalles, chacun étant subdivisé en 4.

A titre d'illustration, voici comment utiliser la commande `plot2d` pour représenter sur un même graphique les fonctions cosinus et sinus sur l'intervalle $[-\pi, \pi]$, avec les caractéristiques suivantes :

i) chaque courbe est représentée par 30 points de discrétisation régulièrement espacés sur l'intervalle proposé.

ii) les courbes cosinus et sinus apparaîssent avec des lignes pleines, respectivement bleu et rouge, et possèdent chacune une légende (par exemple "Fonction cosinus" et "Fonction sinus").

iii) les axes sont gradués.

iv) l'axe des abscisses s'appelle "x", celui des ordonnées "y", et le titre du graphique est par exemple "Voici mon premier graphique Scilab" (on utilisera la fonction `xtitle` pour réaliser cette dernière contrainte).

```
-->xset("window",0)
-->x = linspace(-%pi,%pi,30);
-->xbasc()
```

```
-->plot2d([x x],[cos(x) sin(x)],[10 5],"121","Fonction cosinus@Fonction sinus")
-->xtitle("Voici mon premier graphique Scilab","x","y")
```

Références

[1] G. Chandler et S. Roberts, Introduction to Scilab

[2] B. Pinçon, Une introduction à Scilab, http ://www.iecn.u-nancy.fr/˜pincon

[3] M. Postel, Introduction au logiciel Scilab, http ://www.ann.jussieu.fr/˜postel

[4] B. Ycart, Démarrer en Scilab

Table des matières