Introduction au logiciel Scilab

Marie Postel*

Version révisée janvier 2009

Scilab est un logiciel de calcul numérique développé par l'Institut National de Recherche en Informatique et en Automatique (INRIA) et distribué gratuitement sur presque tout type d'ordinateurs (PC Windows, Linux, Unix, MacIntosh). Pour plus d'information et pour télécharger ce logiciel, vous pouvez consulter le site Internet de l'INRIA :

http ://www.scilab.org.

Scilab possède son propre langage de programmation (très proche de celui de Matlab), de nombreuses fonctions préprogrammées, et des possibilités étendues de visualisation graphique. Il est très simple, très convivial et s'impose de plus en plus comme un outil incontournable dans l'enseignement, la recherche et le développement. C'est un des logiciels utilisés pour l'épreuve d'analyse numérique de l'agrégation de mathématiques. Il permet de réaliser des simulations numériques basées sur des algorithmes d'analyse numérique. Il sera donc utilisé pour la résolution approchée d'équations différentielles, d'équations aux dérivées partielles ou de systèmes linéaires, non linéaires, etc...

Le présent document décrit les rudiments du langage et les commandes indispensables.

*Laboratoire Jacques-Louis Lions, Université Pierre et Marie Curie

Table des matières

1 Fonctionnement général **3**
- 1.1 Première utilisation de `Scilab` . . . 3
- 1.2 La barre de menu horizontale . . . 4
- 1.3 Utilisation de l'aide en ligne . . . 5

2 Type de données **5**
- 2.1 Constantes spéciales . . . 5
- 2.2 Vecteurs ("vectors"...) . . . 6
- 2.3 Matrices ("matrix" ...) . . . 7
- 2.4 Autres objets mathématiques . . . 11

3 Programmation **11**
- 3.1 Edition de scripts . . . 11
- 3.2 Fonctions ou macros ("function" or "macros"...) . . . 11
- 3.3 Les boucles . . . 15
- 3.4 Tests . . . 16
- 3.5 Utilisation de fonctions `Scilab` . . . 16
  - 3.5.1 `fsolve` : systèmes d'équations non linéaires . . . 17
  - 3.5.2 `ode` : solveur d'équations différentielles ordinaires . . . 18

4 Entrées / Sorties sous `Scilab` **20**
- 4.1 Interaction avec l'utilisateur . . . 20
- 4.2 Sauvegarde de résultats en binaire . . . 21
- 4.3 Fichiers ascii . . . 22

5 Sorties graphiques **23**
- 5.1 La fenêtre graphique . . . 23
- 5.2 Les tracés en 2 dimensions . . . 23
- 5.3 Les graphiques dans l'espace . . . 25

1 Fonctionnement général

`Scilab` présente plusieurs fonctionnalités dont un langage de programmation propre, proche dans sa syntaxe des langages informatiques courants (Fortran, C, Pascal) qui est traduit par un interpréteur. Il possède en outre un grand nombre d'algorithmes préprogrammés (dans ce langage ainsi que des librairies de procédures écrites en Fortran et en C) . Enfin depuis la version 2.7 `Scilab` propose un éditeur intégré permettant d'écrire les scripts sans sortir de l'environnement
Une des particularités de `Scilab` qui sera largement exploitée dans ce module est la simplicité de sa syntaxe pour tout ce qui concerne le calcul matriciel. Les opérations élémentaires sur les matrices, comme l'addition, la transposition ou la multiplication sont effectuées immédiatement. D'autres avantages moins spécifiques à l'analyse numérique sont appréciables au niveau de la conception des programmes. `Scilab` permet à l'utilisateur de créer ses propres fonctions et ses librairies très facilement. Les fonctions sont traitées comme des objets et à ce titre peuvent être passées en argument d'autres fonctions. Notons enfin l'interface possible avec des sous-programmes Fortran ou C.
La première partie de cette notice donne le mode d'emploi général de `Scilab`, son utilisation en mode "calculatrice" et détaille les principales fonctionnalités accessibles par les boutons de la fenêtre. La partie 2 présente les principaux types de données et les opérations permettant de les manipuler. La partie 3 s'attaque à des aspects plus complexes de programmation - les `boucles` et les `tests`, qui nécessitent la mise en œuvre de `fonctions` et l'écriture de `scripts`. La partie 4 présente les entrées/sorties sur fichiers et les graphiques font l'objet de la partie 5.

1.1 Première utilisation de `Scilab`

Nous présentons dans ce paragraphe l'utilisation de `Scilab` dans l'environnement WINDOWS. Dans un premier temps, pour se familiariser avec la syntaxe, il est plus facile d'utiliser `Scilab` comme une super calculatrice.
Cliquer sur l'icône `Scilab` (ou `scilex`). Vous allez voir apparaître une nouvelle fenêtre `XScilab` comme sur la figure 1.
En haut de la fenêtre, se trouve une barre de menu :
`File Control Demos Graphic Help Editor`
À l'intérieur de cette fenêtre, on peut rentrer une suite d'instructions sur la ligne matérialisée par une flèche $-->$ qu'exécutera ensuite le logiciel après un retour chariot. Si l'instruction est suivie d'un point virgule, elle est simplement exécutée. Si elle est suivie d'une virgule ou simplement d'un `return / entrée` le résultat apparaît dans la fenêtre. Dans le cas où l'instruction ne comporte pas d'affectation (signe =) à une variable, le résultat apparaît à la suite du mot "an-s" answer=réponse). Les appels à des fonctions graphiques entraînent l'ouverture d'une fenêtre spécifique. Les instructions très longues peuvent être tapées sur plusieurs lignes en utilisant le symbole de continuation ... à la fin de la ligne.

Les valeurs prises par les différentes variables au terme de l'exécution sont conservées en mémoire et il est possible (et parfois utile) d'afficher l'une d'elles a posteriori en tapant simplement son nom. À noter aussi que l'historique des instructions entrées depuis le début de la session sur la

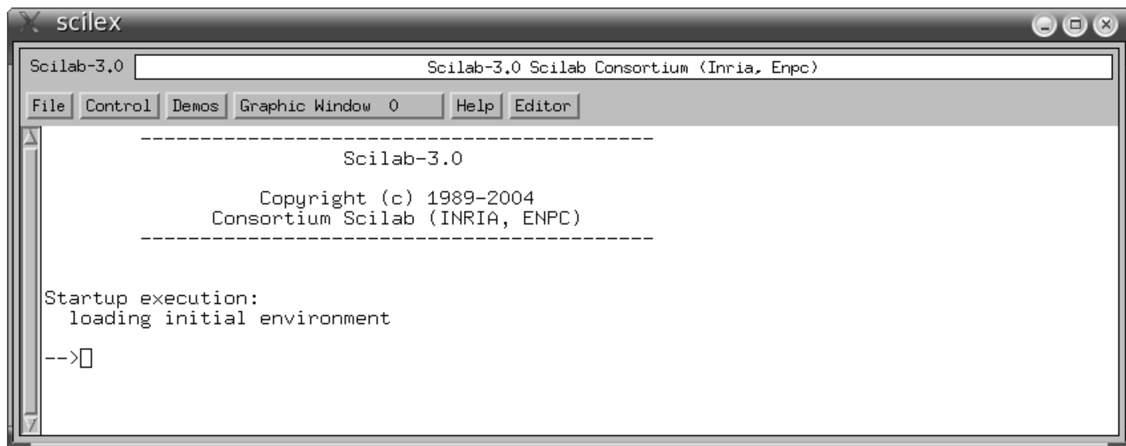

FIG. 1 – Fenêtre de commande `Scilab`

ligne de commande est accessible par pressions successives de la touche ↑. Enfin, pour effacer les données en mémoire (par exemple avant d'exécuter un nouveau calcul), il suffit d'utiliser la commande `clear`.

1.2 La barre de menu horizontale

Bouton **file**. Si vous êtes curieux de voir comment `Scilab` est installé sur votre machine vous pouvez aller visiter les différents sous-répertoires de la distribution par l'intermédiaire du menu `file` puis en cliquant sur `Scilab`. Vous pourrez vous déplacer dans l'arborescence :
• **bin** contient les exécutables.
• **demos** contient les démonstrations. Le script `alldems.dem` permet d'ajouter un nouvel item au menu `demos`.
• **examples** examples de correspondance d'un autre langage à `Scilab`.
• **macros** librairies de fonctions utilisables interactivement.
• **routines** code de ces fonctions.
• **tests** programmes d'évaluation pour tester `Scilab` sur une machine.
• **tmp** programmes écrits par les utilisateurs pendant des cours.
Ce menu `file` vous permet d'accéder également à vos propres fichiers en cliquant sur `Home`. `Scilab` voit les fichiers dans le répertoire courant. Au départ, `home` est celui où vous étiez quand vous avez démarré la session, sous `linux`, ou bien le répertoire de travail par défaut sous `windows`. Les commandes `pwd` et `cd` peuvent être utilisées depuis la fenêtre de commandes pour changer de répertoire.
La barre de menu horizontale en bas de la fenêtre qui s'ouvre quand on clique sur `file` permet de charger des données (`load`) ou des fonctions (`getfc` et `getf`) ou exécuter un script de commandes (`exec`). Nous reviendrons sur ces fonctionnalités au début de la partie Programmation 3.
Bouton **Control**. Ce sous-menu permet d'interrompre les calculs `Scilab` avec `stop`, de les reprendre avec `resume` ou de les abandonner avec `abort`.
Bouton **Demos**. Ouvre un sous-menu permettant de lancer les scripts de présentation du logiciel.

Bouton **Graphic Window**. Ouvre un sous-menu permettant de passer d'une fenêtre graphique à une autre, les créer et les effacer.
Bouton **Help**. Ouvre un sous-menu permettant d'ouvrir la fenêtre d'aide en ligne (`Help browser`) (voir ci-dessous paragraphe 1.3), ou bien à obtenir de l'aide sur une commande dont on connaît le nom (`Apropos`).
Bouton **Editor**. Ouvre une fenêtre d'édition pour écrire les scripts ou les fonctions `Scilab` (voir ci dessous paragraphe 3.1).

1.3 Utilisation de l'aide en ligne

Une aide en ligne sur toutes les commandes `Scilab`, les fonctions préprogrammées, les types de données, etc, est disponible en cliquant sur le bouton `help` puis sur les différents sous-menus. On peut court-circuiter l'arborescence de l'aide en faisant une recherche par mot-clef en cliquant sur la loupe en haut à gauche de la fenêtre d'aide.
Cette aide est rédigée en anglais ; à part ce petit inconvénient, son utilisation est très pratique. En fait il est indispensable d'apprendre rapidement à s'en servir car c'est la seule manière de retrouver les modalités d'utilisation des quelques centaines de fonctions préprogrammées. Noter qu'on peut aussi taper les demandes d'aides au clavier, par exemple

```
help() // donne des informations générales sur l'aide en ligne
apropos motclef // donne toutes les fonctions Scilab qui ont un
//                         rapport avec le mot-clef spécifié
help nomdefonction //ouvre la fenêtre d'aide pour utiliser la
//                               fonction Scilab nomdefonction
```

En cas d'erreur de syntaxe, `Scilab` affiche un message d'erreur juste en dessous de l'instruction fautive avec un point d'exclamation localisant éventuellement l'erreur dans cette instruction. Le message explicatif est parfois laconique - et en anglais - donc il peut être utile de vous constituer un petit lexique des erreurs les plus fréquentes.

2 Type de données

Les différents types scalaires sont les constantes, les booléens, les polynômes, les chaînes de caractères et les rationnels. Avec ces objets on peut définir des matrices. Il existe aussi des listes et des fonctions. Le tableau 1. rassemble les principales fonctions sur les scalaires prévues par `Scilab`

2.1 Constantes spéciales

Elles sont précédées du caractère %. Voici la liste des plus utilisées. La liste complète est obtenue en tapant la commande `who`

$$\%F \quad \%T \quad \%z \quad \%s \quad \%nan \quad \%inf \quad \%t \quad \%f \quad \%eps \quad \%io \quad \%i \quad \%e \quad \%pi$$

2.2 Vecteurs ("vectors"...)

Pour définir un vecteur la syntaxe est une des suivantes :

```
-->v=[2,-3+%i,7]                    //vecteur ligne
 !   2.  - 3. + i      7. !
-->v'                                // vecteur transposé conjugué
 ans  =
!   2.        !
! - 3. - i    !
!   7.        !
-->v.'                                // vecteur transposé
 ans  =
!   2.        !
! - 3. + i    !
!   7.        !
-->w=[-3;-3-%i;2]                   // vecteur colonne
 w  =
! - 3.        !
! - 3. - i    !
!   2.        !
-->v'+w                              //somme de deux vecteurs
 ans  =
! - 1.        !
! - 6. - 2.i !
!   9.        |
-->v*w                              //produit scalaire euclidien
 ans  =
    18.
-->w'.*v    // produit des composantes (essayer aussi avec ./)
 ans  =
! - 6.    8. - 6.i    14. !
```

Les composantes sont séparées par des blancs (dangereux) ou des virgules pour les vecteurs lignes et par des points-virgules pour les vecteurs colonnes. Des messages erreurs sont affichées si une opération impossible est tentée (par exemple l'addition de vecteurs de longueurs différentes).
Et pour aller plus vite...

```
-->v=1:5.1:23              //vecteur à incrément constant
 v  =
!   1.    6.1    11.2    16.3    21.4 !
-->ones(v)
 ans  =
```

```
!   1.    1.    1.    1.    1. !
-->ones(1:4)
 ans  =
!   1.    1.    1.    1. !
-->3*ones(1:5)
 ans  =
!   3.    3.    3.    3.    3. !
-->zeros(1:3)
 ans  =
!   0.    0.    0. !
```

2.3 Matrices (“matrix” ...)

Les matrices suivent la même syntaxe que les vecteurs. Les composantes des lignes sont séparées par des virgules et chaque ligne est séparée de l’autre par un point virgule.

```
-->// une manière de définir une matrice 3 x 3:
-->A=[1,2,3;0,0,atan(1);5,9,-1];

-->// une autre syntaxe pour faire la même chose
-->A=[1 2 3
-->     0 0 atan(1)
-->     5 9 -1]
 A  =
!   1.    2.    3.        !
!   0.    0.     .7853982 !
!   5.    9.  - 1.        !

-->// à ne pas confondre avec ce groupe d’instructions
-->A=[1 2 3...
-->0 0 atan(1)...
-->5 9 -1]
 A  =
!   1.    2.    30.    0.    0.7853982    5.    9.  - 1. !

-->
-->v=1:5;W=v'*v           // multiplication de matrices
 W  =
!   1.    2.     3.     4.     5.  !
!   2.    4.     6.     8.     10. !
!   3.    6.     9.     12.    15. !
!   4.    8.     12.    16.    20. !
```

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	arccos
asin	arcsin
atan	arctangente
cosh	ch, cosinus hyperbolique
sinh	sh, sinus hyperbolique
tanh	th, tangente hyperbolique
acosh	argch
asinh	argsh
atanh	argth
sqrt	racine carrée
floor	partie entière ($E(x) \leq x < E(x) + 1$)
ceil	partie entière supérieure ($n < x \leq E(x) + 1$)
int	partie entière anglaise (floor si $x > 0$ et ceil sinon)
round	arrondi ($E(x + 0.5)$)
erf	fonction erreur $\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x \exp(-t^2)dt$
erfc	complémentaire $\text{erfc}(x) = 1 - \text{erf}(x)$
gamma	$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} dt$
gammaln	$\ln(\Gamma(x))$
dlgamma	$\frac{d}{dx} \ln(\Gamma(x))$

TAB. 1 – Fonctions élémentaires sur les scalaires.

```
!   5.    10.    15.    20.    25. !
-->
-->W(1,:)            //extraction de  la première ligne
 ans  =
!   1.    2.    3.    4.    5. !
-->
-->W(:,$)        //extraction de la dernière colonne  $
  ans  =
!   5.  !
!   10. !
!   15. !
!   20. !
!   25. !
-->
-->A=eye(3,3)     // Matrice identité
 A  =
!   1.    0.    0. !
!   0.    1.    0. !
!   0.    0.    1. !
-->
-->B=toeplitz([2,1,0,0])   // Matrice de Toeplitz symétrique
 B  =
!   2.    1.    0.    0. !
!   1.    2.    1.    0. !
!   0.    1.    2.    1. !
!   0.    0.    1.    2. !
```

Le tableau 2. résume les principales fonctions affectant ou effectuant des opérations sur des matrices. Noter que les fonctions du tableau 1. peuvent aussi s'appliquer à des matrices, composante par composante, comme dans l'exemple suivant

```
-->u=[0:1:4]
 u  =
!   0.    1.    2.    3.    4. !

-->v=sin(u)
 v  =
!   0.     .8414710     .9092974     .1411200  -  .7568025 !
```

Fonction	Description
ones(i,j)	crée un tableau de i lignes j colonnes contenant des 1
zeros(i,j)	crée un tableau de i lignes j colonnes contenant des 0
eye(i,j)	crée un tableau de i lignes j colonnes avec des 1 sur la diagonale principale et 0 ailleurs
toeplitz(u)	crée une matrice de Toeplitz symétrique dont la première ligne est le vecteur u
diag(u)	crée une matrice carrée avec le vecteur u sur la diagonale et 0 ailleurs
diag(U)	extrait la diagonale de la matrice U
triu(A)	renvoie la partie supérieure de A
tril(A)	renvoie la partie inférieure de A
linspace(a,b,n)	crée un vecteur de n composantes uniformément réparties de a à b
linsolve(A,b)	résolution du système linéaire `Au+b=0`
A\b	résolution du système linéaire `Au=b`
cond(A)	conditionnement d'une matrice (norme euclidienne)
det(A)	déterminant d'une matrice
rank(A)	rang d'une matrice
inv(A)	inverse d'une matrice
pinv(A)	pseudo inverse d'une matrice
svd(A)	valeurs singulières d'une matrice
norm(A)	norme matricielle ou vectorielle
u'	prend le transposé conjugué de u
u.'	prend le transposé conjugué de u
u*v	multiplication matricielle
u+v	addition matricielle
u-v	soustraction matricielle
u.* v	multiplication des tableaux u et v terme à terme
u./v	division du tableau u par le tableau v terme à terme

TAB. 2 – Principales opérations sur les matrices.

2.4 Autres objets mathématiques

Un grand nombre d'objets mathématiques ou logiques peut intervenir dans un programme de simulation (entier, réel, fonction, polynôme, matrice, booléen, etc...). Il est donc indispensable de connaître la manière d'affecter puis de manipuler chacun d'eux : Le tableau 3 dresse un récapitulatif de la syntaxe à respecter. La démonstration `Introduction to Scilab` permet également de faire défiler en mode pause différents exemples d'affectation et d'utilisation de ces objets (il est nécessaire de faire un retour chariot pour que le programme exécute chaque nouvelle ligne).

3 Programmation

3.1 Edition de scripts

Dès que le calcul à effectuer implique un enchaînement de commandes un peu compliqué, il vaut mieux écrire un `script`.
Si vous avez installé la dernière version du logiciel `Scilab`, vous avez aussi à votre disposition un éditeur intégré qu'on lance en cliquant sur le bouton `Editor`. Cet éditeur gère les suffixes des fichiers que vous créez : s'il s'agit d'une fonction qui se termine par la commande `EndFunction`, le fichier prendra automatiquement le suffixe `sci`, dans le cas contraire, c'est un `script`, un enchaînement de commandes `Scilab`, et il prendra le suffixe `sce`. Le bouton `Execute` dans la fenêtre d'édition permet de lancer l'exécution du script courant dans la fenêtre de commandes (`Load into Scilab`). Les fonctions, si elles sont définies dans un fichier indépendant, doivent d'abord être chargées (avec la commande `getf`) et utilisées avec la syntaxe d'appel détaillée dans le paragraphe 3.2.
Nous allons maintenant détailler quelques règles de programmation qui pour la plupart ne devraient pas surprendre outre mesure si on connaît déjà un autre langage (C ou Fortran).

3.2 Fonctions ou macros ("function" or "macros"...)

Les fonctions sont des enchaînements de commandes `Scilab` qui sont exécutées à part. Elles peuvent être créées et exécutées de plusieurs manières. Elles peuvent avoir des arguments d'appel, de retour, inclure des tests conditionnels, des boucles et s'appeler récursivement. Enfin elles peuvent elles-mêmes être des arguments de fonctions.
Une fonction ne comportant qu'un petit nombre d'instructions peut être définie directement dans la fenêtre de commandes de la manière suivante

```
-->deff('[s]=angle(x,y)','s=180*atan(y/x)/%pi')
-->angle(4,5)
 ans  =
    51.340192
```

La fonction `angle` donne la mesure en degré de l'angle entre le segment formé par l'origine et le point de coordonnées (x, y) et l'axe des abscisses.

Instruction	Description
`a=2`	affectation de la valeur réelle 2 à `a`
`a=%t`	affectation du booléen *vrai* à `a`
`p=poly([%i -%i],'z')`	affectation du polynôme $(z-i)(z+i)=z^2+1$ à `p`
`z=poly(0,'z');`	
`q=1+3*z+4.5*z^2`	affectation du polynôme $1+3z+4.5z^2$ à `q`
`r=p/q`	affectation d'une fraction rationnelle à `r`
`A=[a+1 2 ; atan(1) -3]`	affectation d'une matrice réelle 2×2 à `A`
`rand(3,3,'u')`	matrice aléatoire 3×3 (loi uniforme sur $[0,1]$)
`rand(3,3,'n')`	matrice aléatoire 3×3 (loi gaussienne centrée)
`eye(4,4)`	matrice identité $I_4 \in \mathcal{M}_4(I\!R)$
`ones(1,2);1 :2 :7`	vecteurs lignes $(1,1)$ et $(1,3,5,7)$
`B=toeplitz([4 1 0 0])`	matrice tridiagonale symétrique dans $\mathcal{M}_4(I\!R)$
`C=B'*B`	calcul de $C={}^tBB$ (matrice 4×4)
`D=C.*B`	multiplication terme à terme de `C` par `B`
`f=C(1 :4,1)` ou `f=C( :,1)`	extraction des termes de la première colonne de `C`
g=$D\backslash$f	division à gauche de `f` par `D` (Dg=f)
`deff('y=f(a,b)','y=a+b')`	définition la fonction f(a,b)=a+b

TAB. 3 – Instructions syntaxiques et commandes élémentaires.

Dès que la fonction nécessite plusieurs instructions, il vaut mieux définir les fonctions dans un fichier à part à l'aide de l'éditeur de texte. Pour l'exemple précédent on peut créer le fichier "angle.sci" contenant les lignes :

```
function [s]=angle(x,y)
s=180*atan(y/x)/%pi;                // Calcule  l'angle en degrés
xpoly([x,0,x],[0,0,y],"lines")     // Visualise le triangle
endfunction
```

puis dans la fenêtre de commandes on tape

```
getf("angle.sci","c");
```

et on peut maintenant utiliser la fonction "angle" comme précédemment. En plus de la mesure de l'angle en degré, la fonction affiche graphiquement le triangle formé par l'origine et les points de coordonnées $(x, 0)$ et (x, y). Dans l'exemple ci-dessus le paramètre $''c''$ dans l'appel à `getf` est optionnel. Il requiert la compilation des fonctions contenues dans le fichier toto.sci au moment de leur chargement.

Il existe de nombreux algorithmes préprogrammés dans `Scilab` pouvant être utilisés dans des programmes de simulation plus complexes comme "boite noire". Ils sont chargées automatiquement au lancement de la session. Tous sont répertoriés et brièvement présentés dans l'aide en ligne. Il est très utile de connaître au moins le nom des instructions associées à ces derniers (au nombre d'une quinzaine) sans retenir forcément leur syntaxe complète, facilement retrouvable avec l'aide en ligne (ou l'instruction `help`).

De manière générale, la syntaxe de définition d'une fonction externe est

```
function [y_1,...,y_m]=toto(x_1,....,x_n)
..
..
..
endfunction
```

où `toto` est le nom de la fonction, $x_1, ...x_n$, les `n` arguments d'entrée et $[y_1, ...y_m]$ les `m` arguments de sortie. Reprenons l'exemple précédent de la fonction angle, qui calcule et dessine l'angle formé par le segment d'extrémités `(0,0)` et `(x,y)` avec l'horizontale.
Le passage des arguments d'entrée dans les fonctions se fait par valeur. Aussi, même si elles sont modifiées dans la fonction les valeurs des paramètres ne sont pas modifiées dans le programme appelant.
Si une des variables de la procédure n'est pas définie à l'intérieur de celle-ci et non plus en argument d'entrée, elle prend la valeur définie dans le programme appelant. Ceci permet d'appeler les fonctions avec moins de paramètres d'entrée que prévu. Ainsi

```
-->clear
-->getf(``angle.sci'')
-->angle(4)
 !--error     4
undefined variable : y
at line       2 of function angle                    called by :
angle(4)
```

donne une erreur car le paramètre `y` n'est pas affecté. En revanche

```
-->clear
-->getf(``angle.sci'')
-->y=2
 y  =
    2.
-->angle(4)
 ans  =
    1.1071487
-->
```

s'exécute correctement en utilisant pour le paramètre `y` la valeur connue dans le script principal. La récupération des valeurs calculées par la fonction se fait par les paramètres de sortie ($[y_1, ...y_n]$ dans la définition de `toto` ci-dessus). S'il n'y a qu'un résultat comme dans l'exemple de la fonction `angle`, on peut se dispenser de le récupérer dans une variable, en revanche s'il y a plus d'un paramètre de sortie, il faut récupérer leurs valeurs dans des variables dans le script d'appel. Regardons par exemple l'utilisation de la fonction `polaire` définie ci-dessous et sauvée dans le fichier `polaire.sci`

```
function [r,theta]=polaire(x,y)
r=sqrt(x^2+y^2)
theta=atan(y,x)
```

Pour l'utiliser à partir de la fenêtre `Scilab` on tape les instructions suivantes

```
-->;getf("polaire.sci");

-->polaire(2,3)    // Si seulement  le rayon nous intéresse
 ans  =
    3.6055513

-->// Si on veut récupérer à la fois le rayon et l'angle
```

```
-->[r,t]=polaire(2,3)
 t  =
     .9827937
 r  =
    3.6055513
```

Enfin pour clore ce paragraphe d'introduction à la programmation sous `Scilab`, notez que certaines commandes spéciales ne peuvent s'utiliser que dans une fonction :
`argn` : donne le nombre de paramètres d'entrée et de sortie de la fonction
`error` : interrompt l'exécution de la fonction, affiche le message d'erreur et retourne dans le programme appelant.
`warning` : imprime le message mais ne retourne pas dans le programme appelant
`pause` : interrompt l'exécution jusqu'à ce que l'utilisateur tape un `return`
`break` : sort d'une boucle.
`return` : retourne dans le programme appelant et renvoie des valeurs locales.
Depuis la version 2.7 les fonctions peuvent être définies à l'intérieur du script appelant, avant d'être utilisées. Elles doivent dans ce cas se terminer par l'instruction `endfunction`, *qui n'est pas obligatoire si elles sont définies dans un fichier de fonctions externes.*

3.3 Les boucles

Il y en a de deux types en `Scilab` : les boucles `while` et les boucles `for`. La boucle `for` parcourt un vecteur d'indices et effectue à chaque pas toutes les instructions délimitées par l'instruction `end`.

```
-->x=1; for k=1:4,x=x*k, end
 x  =
    1.
 x  =
    2.
 x  =
    6.
 x  =
    24.
```

la boucle `for` peut parcourir un vecteur (ou une matrice) en prenant comme valeur à chaque pas les éléments (ou les colonnes) successifs.

```
-->v=[-1 3 0]
 v  =
! - 1.     3.     0. !

-->x=1; for k=v, x=x+k, end
```

```
 x  =
    0.
 x  =
    3.
 x  =
    3.
```

La boucle `while` effectue une suite de commandes tant qu'une condition est satisfaite.

```
-->x=1; while x<14,x=x+5,end
 x  =
    6.
 x  =
    11.
 x  =
    16.
```

Les deux types de boucles peuvent être interrompus par l'instruction `break`. Dans les boucles imbriquées `break` n'interrompt que la boucle la plus interne.

3.4 Tests

Dans `Scilab` on dispose du classique `if-then-else` agrémenté du `elseif` parfois bien utile. La syntaxe est par exemple

```
-->x=16
 x  =
    16.

-->if x>0 then, y=-x, else y=x, end
 y  =
  - 16.
```

Dans le cas où le test doit départager un "grand nombre" (supérieur à deux...) de possibilités, on a aussi la possibilité d'utiliser le `select-case`, équivalent du `switch-case` en C.

```
-->n=round(10*rand(1,1))
-->select n
-->case 0 then
-->    printf('cas numero 0')
-->case 1 then
-->    printf('cas numero 1')
-->else
-->   printf('autre cas')
-->end
```

français	anglais	test Scilab
et	and	&
ou	or	|
non	not	~
égal	equal	==
différent	different	<> ou ~=
plus petit que	lower than	<
plus grand que	greater than	>
plus petit ou égal à	lower than or equal	<=
plus grand ou égal à	larger than or equal	>=

TAB. 4 – Les opérateurs logiques dans les tests.

3.5 Utilisation de fonctions Scilab

Dans ce paragraphe nous détaillons l'utilisation de deux fonctions Scilab, fsolve qui sert à résoudre des systèmes d'équations non linéaires et ode pour résoudre des équations différentielles ordinaires. La description exhaustive de ces deux programmes est disponible en anglais dans l'aide en ligne

3.5.1 fsolve : systèmes d'équations non linéaires

La fonction fsolve calcule les solutions de systèmes d'équations non linéaires. La syntaxe d'appel la plus simple est
`x=fsolve(x0,fct)`
et peut être précisée avec des arguments optionnels :
`[x [,v [,info]]]=fsolve(x0,fct [,fjac] [,tol])`.
Ces paramètres d'appels et de sortie sont :
`x0` : un vecteur de réels (valeurs initiales de l'argument de la fonction).
`fct` : une fonction externe.
`fjac` : une fonction externe.
`tol` : un scalaire réel pour la tolérance sur l'erreur : la recherche des racines s'arrête quand l'algorithme estime que l'erreur entre les solutions et x est majorée par tol (La valeur par défaut est tol=1.d-10).
`x` : vecteur de réels contenant la solution du système.
`v` : vecteur de réels contenant la valeur de la fonction en x.
`info` : indicateur de fin :
0 : mauvais paramètres d'entrée.
1 : calcul correct avec une erreur inférieure à la tolérance tol.
2 : nombre maximum d'appels à la fonction atteint.
3 : tol est trop petit, l'algorithme ne peut plus progresser.
4 : l'algorithme ne progresse plus.

Description
La fonction `fsolve` trouve les solutions d'un système d'équations non linéaires donné sous la forme
`0 = fct(x)`
`fct` est une fonction externe dont la syntaxe d'appel est `v=fct(x)`.
`jac` est aussi une fonction externe qui renvoie le jacobien de `fct` c'est-à-dire `v=d(fct)/dx(x)`

Exemples

```
// Un exemple simple : système non linéaire
deff('[y]=fsol1(x)',...
       'y=[x(1)^2+2*x(1)*x(2)+1; 2*x(1)*x(2)+x(2)^2+2]');
deff('[y]=fsolj1(x)',...
     'y=[2*x(1)+2*x(2), 2*x(1) ; 2*x(2), 2*x(1)+2*x(2)]');
[xres]=fsolve([100;100],fsol1)
fsol1(xres)
[xres]=fsolve([100;100],fsol1,fsolj1)
fsol1(xres)
```

Les deux méthodes donnent le même résultat, mais la première laisse le programme évaluer le jacobien de la fonction par différences finies alors que la deuxième lui fournit la fonction `fsol1` pour calculer les dérivées partielles. Dans ce cas simple les performances en temps de calcul sont légèrement en faveur de la deuxième méthode.

3.5.2 `ode` : solveur d'équations différentielles ordinaires

Le nom de ce programme vient de l'acronyme anglais *ode - ordinary differential equation solver* .

La syntaxe d'appel la plus simple est
`y=ode(y0,t0,t,f)` où `y0` est le vecteur des conditions initiales, `t0` est le temps initial, `t` est le vecteur des temps où on veut calculer la solution et `y` est la matrice contenant la solution `y=[y(t(1)),y(t(2)),...]` .
L'argument `f` de `ode` est une fonction externe dont la syntaxe d'appel doit être la suivante
`ydot = f(t,y)`
où `t` est un scalaire réel (le temps) et `y` un vecteur de réels. Cette fonction est le second membre du système différentiel $dy/dt = f(t, y)$.
L'argument `f` peut aussi être une *liste* `Scilab`, avec la structure `lst=list(f,u1,u2,...un)`, où `f` est alors une fonction avec comme syntaxe d'appel :
`ydot = f(t,y,u1,u2,...,un)`
ce qui permet d'utiliser des paramètres en argument de `f`.

Des syntaxes d'appel avec des paramètres optionnels d'appel et de sortie sont possibles

```
[y,w,iw]=ode([type],y0,t0,t [,rtol [,atol]],f [,jac])
[y,rd,w,iw]=ode("root",y0,t0,t [,rtol [,atol]],f [,jac])
```

Les paramètres supplémentaires sont :

`type` : une des chaînes de caractères suivantes : "adams" "stiff" "rk" "rkf" "fix" "discrete" "roots"
`ode` est l'interface `Scilab` pour utiliser différents solveurs d'équations différentielles, en particulier `ODEPACK`. le type de problème et la méthode utilisée pour le résoudre sont définis dans le premier argument (optionnel). Par défaut, le solveur `lsoda` de ODEPACK est utilisé. Ce solveur sélectionne automatiquement la méthode la plus adaptée, parmi les méthodes de prediction-correction d'Adams et la méthode BDF (Backward Differentiation Formula).
Si le premier paramètre `type` est précisé, il peut s'agir par exemple de :
"adams" : Pour les problèmes non raides.
"stiff" : Pour les problèmes raides.
"rk" : Méthode de Runge-Kutta adaptative d'ordre 4.

`rtol,atol` : vecteurs de constantes réelles vectors de la même taille que y .
Des paramètres optionnels peuvent préciser l'erreur tolérée sur la solution `rtol` et `atol` sont les seuils pour les erreurs relative et absolue.

`jac` : fonction externe .
Pour des problèmes raides, il est préférable de fournie à `ode` le jacobien de la fonction second membre, sous la forme d'une fonction externe dont la syntaxe d'appel sera
`J=jac(t,y)`
La matrice `J` contient les valeurs de `df/dy` en `(t,y)`, c'est-à-dire `J(k,i) = dfk /dyi` où `fk` est la k^{ieme} composante de `f` et `yi` est la i^{ieme} composante de `y`.

Exemples

```
// Equation différentielle scalaire
// dy/dt=y^2-y sin(t)+cos(t), y(0)=0
deff("[ydot]=f(t,y)","ydot=y^2-y*sin(t)+cos(t)")
y0=0;t0=0;t=0:0.1:%pi;
y=ode(y0,t0,t,f);
plot(t,y)
// Simulation de dx/dt = A x(t) + B u(t) avec u(t)=sin(omega*t),
// x0=[1;0]
// solution x(t) désirée à t=0.1, 0.2, 0.5 ,1.
// La matrice A et la fonction u sont passées à la fonction
//    second membre dans une liste.
// La matrice B et  omega sont passés comme des variables globales
```

```
deff("[xdot]=linear(t,x,A,u)","xdot=A*x+B*u(t)")
deff("[ut]=u(t)","ut=sin(omega*t)")
A=[1 1;0 2];B=[1;1];omega=5;
ode([1;0],0,[0.1,0.2,0.5,1],list(linear,A,u));
//
// Notation matricielle
// Intégration de l'équation différentielle de  Riccati
// Xdot=A'*X + X*A - X'*B*X + C , X(0)=Identité
// Solution a t=[1,2]
deff("[Xdot]=ric(t,X)","Xdot=A''*X+X*A-X''*B*X+C")
A=[1,1;0,2]; B=[1,0;0,1]; C=[1,0;0,1];
t0=0;t=0:0.1:%pi;
X=ode(eye(A),0,t,ric)
//
// Calcul de exp(A) comme solution de dy/dt=Ay, avec y(0)=y0
// La solution y(t)=exp(At)y0
A=[1,1;0,2];
e1=[1;0];e2=[0;1];
deff("[xdot]=f(t,x)","xdot=A*x");
[ode(e1,0,1,f),ode(e2,0,1,f)]
[ode("adams",e1,0,1,f),ode("adams",e2,0,1,f)]
// valeur exacte
expm(A)
// Avec une matrice raide, en donnant le Jacobien
A=[10,0;0,-1];
deff("[xdot]=f(t,x)","xdot=A*x");
deff("[J]=Jacobian(t,y)","J=A")
[ode("stiff",e1,0,1,f,Jacobian),ode("stiff",e2,0,1,f,Jacobian)]
// valeur exacte
expm(A)
```

4 Entrées / Sorties sous `Scilab`

4.1 Interaction avec l'utilisateur

Pour communiquer avec l'utilisateur du programme `Scilab` on peut utiliser la commande `input` dont la syntaxe est :
`[x]=input(``message'', [``string''])`
``` ``message'' ``` est une chaîne de caractères qui s'affiche dans la fenêtre de commandes.
`x` est la variable où on va stocker la valeur entrée par l'utilisateur en réponse à cette question. Ce sera une variable de type réel, sauf si le mot-clef ``` ``string'' ``` était spécifié en argument.
```

Pour afficher des résultats avec un format on utilise la commande
`printf(‘‘format’’,variable)`
comme dans les exemples ci-dessous

```
-->a=1/3;

-->printf(" valeur scalaire =%f",a);
 valeur scalaire =0.333333
-->a=2;

-->printf(" valeur entière =%d",a);
 valeur entière =2
-->a="une chaîne de caractères";

-->printf(" pour afficher %s",a);
 pour afficher une chaîne de caractères
-->b=1;

-->a="chaîne de caractères";

-->printf(" pour afficher %d %s",b,a);
 pour afficher 1 chaîne de caractères
```

Des “interfaces utilisateurs” plus élaborées peuvent être programmées avec des menus. L’aide en ligne sur les commandes `uimenu, addmenu, etc.` donne quelques exemples pour démarrer.

4.2 Sauvegarde de résultats en binaire

A un moment donné, une session `Scilab`dispose d’un certain nombre de données et de fonctions. Nous allons maintenant voir les différents moyens de sauvegarder et de récupérer ces données.

`who` : Pour en avoir la liste
`save` : Pour sauver des variables dans un fichier.
`load` : Pour récupérer des variables précédemment sauvées par `save` .
`clear` : Pour effacer toutes ou une partie des variables.
l’ exemple suivant de sauvegarde d’un vecteur illustre l’emploi des trois dernières fonctions

```
-->a=[1:2:10]                  //définition  du tableau a
 a  =
!   1.    3.    5.    7.    9. !

-->// sauvegarde dans le fichier binaire 'tab_a'
-->save('tab_a',a)
```

```
-->clear a          // effacement du tableau a

-->a                // a n'existe plus
  !--error     4
undefined variable : a


-->// récupération du tableau a sauvé dans 'tab_a'
-->load('tab_a')

-->a
 a  =
!   1.    3.    5.    7.    9. !
```

On peut sauvegarder, effacer, et récupérer l'ensemble d'une session en ne précisant aucune variable dans l'appel de `save, clear` et `load`.
Attention ! les données sauvées de cette manière ne peuvent être relues que par `Scilab`. Les fichiers sont binaires et illisibles avec un éditeur de texte. Pour faire communiquer des données avec un autre logiciel ou avec un éditeur il faut utiliser des fichiers `ascii`.

4.3 Fichiers ascii

On peut lire et écrire dans des fichiers `ascii` avec les commandes `read` et `write` dont la syntaxe est similaire à leurs homonymes en Fortran. Voyons dans l'exemple suivant comment relire les données stockées dans un fichier ascii "fichier.dat" contenant les lignes suivantes

```
3
1 2 3 4
5 6 7 8
9 10 11 12
```

La relecture se fera avec les instructions suivantes :

```
-->Nomfichier='fichier.dat'
-->fid = file('open',Nomfich,'old');
-->// Lecture du nombre de lignes sur le fichier
-->N =read(fid,1,1);
-->// Lecture des N lignes contenant chacune 4 valeurs
-->// dans un tableau tmp
tmp = read(fid,N,4);
```

A l'inverse pour écrire en "format libre" dans un fichier plusieurs enregistrements successifs, on peut procéder de la manière suivante

```
N=3,M=4
tmp=rand(N,M)
fid = file('open','NouveauFichier','new');
write(fid,N)
write(fid,tmp)
file ('close',fid)
```

5 Sorties graphiques

5.1 La fenêtre graphique

A l'aide du bouton `Graphic Window x`, on peut ouvrir plusieurs fenêtres graphiques numérotées `ScilabGraphicx x`. Elles sont activées seulement une à la fois. Elles ont une barre de menu comprenant quatre boutons :

- `3D Rot` : sert à positionner le point de vue à l'aide de la souris dans le cas de graphiques 3D uniquement. Les coordonnées sphériques en cours sont indiquées en haut de la fenêtre.
- `2D Zoom` : peut être appelé plusieurs fois de suite.
- `Unzoom` : revient au graphique d'origine.
- `File` : permet d'accéder au sous-menu suivant
  - `Clear` : efface le graphique
  - `Print` : imprime directement vers une imprimante à sélectionner.
  - `Export` : sauve le graphique dans un format utilisable en dehors de `Scilab` (Postscript, Postscript-Latex, Xfig)
  - `Save` : sauve le graphique dans un fichier dans le format `Scilab`. On pourra le récupérer, toujours sous `Scilab`, par la commande `load`.
  - `Close` : ferme la fenêtre.

Les paramètres de graphique peuvent être fixés par des paramètres optionnels dans les appels aux différentes fonctions graphiques ou bien indépendamment par la fonction `xset`

5.2 Les tracés en 2 dimensions

• `plot2d` C'est la procédure de base pour représenter une courbe dans un plan.

plot2d(x,y,[style,chaîne,légende,rectangle]) On ne détaille ici que les principales options. (Consulter l'aide en ligne pour une liste exhaustive.)

- `x, y` sont les matrices de taille `[npt,nc]` contenant les abcisses et les ordonnées. `nc` est le nombre de courbes et *npt* le nombre de points - le même pour toutes les courbes.
- `style` est un vecteur de dimension `(1,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)`

– `chaîne` est une chaîne de trois caractères `"xyz"`
  si `x=1` la légende est affichée, elle est fournie dans l'argument `legende` sous la forme `"legende1@legende2@...."`
  `y` gère le cadrage :
  si `y=0` les bornes courantes sont utilisées.
  si `y=1` l'argument `rectangle` est utilisé pour redéfinir les bornes `rectangle=[xmin,ymin,xmax,ymax]`
  si `y=2` les bornes sont calculées à partir des valeurs extrèmes de `x` et `y`
  `z` gère l'entourage du graphique :
  `z=1` pour avoir des axes gradués
  $z = 2$ pour avoir un cadre autour du graphique

• Trois exemples très simples :
Dans le premier, `plot2d` trace la courbe *y(x)* en calculant automatiquement le repère.

```
x=[0:1:100];
y=sin(2*%pi*x/100);
z=3*y;
plot2d (x,y,[1],'121','y(x)');
```

Dans le deuxième, le repère est imposé par les valeurs de `cadre` :

```
xbasc()
cadre=[min(x),min(y),max(x),max(y)];
plot2d(x,y,[1],'111','y(x)',cadre);
```

Ceci permet, comme dans le troisième exemple, de tracer plusieurs courbes dans le même repère d'axes, avec des appels à `plot2d` différents

```
xbasc()
cadre = [min(x),min(min(y),min(z)),max(x),max(max(y),max(z))]
plot2d(x,y,[1,1],'111','y(x)',cadre);
plot2d (x,z,[2,2],'111','z(x)',cadre);
```

Ici la première courbe apparait en noir avec sa légende en position 1 - (3eme argument `[1,1]`) et la deuxième courbe est en bleu avec sa légende en position 2 (3eme argument `[2,2]`)
L'appel `xbasc()`, efface la fenêtre graphique sans la fermer. On obtient le même résultat avec `clear` du menu `file` de la fenêtre graphique.

• `fplot2d` est utilisée pour représenter des courbes définies par un tableau d'abcisses et une fonction.

```
deff('y=f(x)','y=sin(x)')
fplot2d(0:%pi/30:%pi,f)
```

• `champ` ou `champ1` pour représenter un champ de vecteurs.
• `grayplot` représentation bidimensionnelle d'une matrice par des niveaux de gris.

```
t=-%pi:%pi/50:%pi;
m=sin(t)'*cos(t);grayplot(t,t,m);
```

Pour les formes géométriques

• `xsegs` trace un ensemble de segments non reliés.
• `xrect` trace un rectangle.
• `xfrect` remplit un rectangle.
• `xrects` dessine et/ou remplit un ensemble de rectangles.
• `xpoly` dessine une ligne brisée.
• `xpolys` dessine un ensemble de lignes brisées.
• `xfpoly` remplit un polygone.
• `xfpolys` remplit un ensemble de polygones.
• `xarrows` dessine un ensemble de flèches

Pour les chaines de caractères

• `xstring` écrit une chaine de caractères
• `xstringl, xstringb` pour encadrer une chaine de caractères avec un rectangle.
• `xnumb` écrit des nombres.

5.3 Les graphiques dans l'espace

• `plot3d` pour représenter graphiquement une matrice à trois dimensions :
Dans l'appel le plus simple, `plot3d(x,y,z)`. `x, y` et `z` sont trois matrices et `z` contient les valeurs des points de coordonnées `(x,y)`.
• `param3d` pour représenter des courbes paramétrées en trois dimensions.
• `contour` pour les courbes de niveaux d'une fonction 3d donnée par une matrice.
• `fec` pour les courbes de niveaux d'une fonction donnée par ses valeurs nodales sur un maillage triangulaire.

Pour conclure ce paragraphe, si vous êtes très satisfaits de vos calculs et de leur représentation graphique, vous aurez envie de les imprimer. On peut le faire par le menu `file`, puis le sous-menu `print` de la fenêtre graphique. On peut aussi sauver le graphique dans un fichier `postscript`, par le sous menu `export` de `file`, ce qui permet de l'imprimer ou de le visualiser ultérieurement et aussi de l'insérer dans un rapport.

Instruction	Description
`plot2d(x,y)`	tracé de la courbe passant par les points `(x,y)`
`plot2d1('oll',x,y)`	idem avec échelle logarithmique sur les deux axes
`fplot2d(x,f)`	tracé de la courbe `(x,f(x))`
`champ(x,y,fx,fy)`	tracé du champ de vecteurs `(fx(x,y),fy(x,y))`
`plot3d(x,y,z)`	tracé de la surface passant par les points `(x,y,z)`
`param3d(x,y,z)`	tracé de la courbe paramétrée passant en `(x,y,z)`
`contour(x,y,z,n)`	tracé de `n` courbes de niveau d'une surface
`histplot(n,data)`	histogramme de l'échantillon `data` divisé en `n` classes
`xbasc()`	effacement des fenêtres graphiques
`xset()`	modification des options graphiques
`xsetech([x1,y1,x2,y2])`	découpage d'une fenêtre graphique
`xstring(x,y,'coucou')`	inscription de caractères sur une figure

TAB. 5 – Principales instructions graphiques

Bibliographie

Documentation en français disponible sur internet :

• Site `Scilab` : http ://www.scilab.org
• Ce document en format Postscript et PDF : http ://www.ann.jussieu.fr/˜postel
• Le support du cours `Scilab` LM206 http ://www.ann.jussieu.fr/ dumas/lm206.html
• Un cours `Matlab` http ://www.ann.jussieu.fr/ postel/matlab/
• Des exemples de scripts `Scilab` pour les équations différentielles ordinaires http ://www.ann.jussieu.fr/ postel/EDO/

Des livres

• Modélisation à l'oral de l'Aggrégation. Calcul Scientifique. Laurent Dumas, collection CAPES / Agrégation, Ellipses, Paris (2000).
• Introduction à `Scilab`. Exercices pratiques corrigés d'algèbre linéaire. Allaire, G., Kaber, S.M., Ellipses, Paris (2002).
• Cours d'Algèbre Linéaire Numérique. Allaire, G., Kaber, S.M. Ellipses, Paris (2002).
• Méthodes d'approximation - Équations différentielles - Applications Scilab. Niveau L3. Guerre-Delabrière, S., Postel, M. Ellipses, Paris (2004).

Merci d'envoyer vos remarques et suggestions sur ce document par email à

postel@ann.jussieu.fr