Chapitre 2

Introduction à Scilab

Sommaire

2.1 Aperçu et environnement

Scilab est un langage de programmation associé à une riche collection d'algorithmes numériques qui couvrent de nombreux aspects des problèmes de calcul scientifique. C'est un pseudo-clone libre de Matlab qui tourne sous Windows, Linux et Mac.

Développé par l'INRIA (aujourd'hui Consortium Scilab-Digiteo).

Plus précisément, c'est un langage interprété (du style LISP, CAML, ...). Cela permet de faire des développements rapides grâce à des fonctionnalités de haut niveau.

Interfaçage avec des langages comme Fortran ou C, mais aussi LabVIEW.

Plus précisément, Scilab possède des méthodes usuelles de haut niveau traitant de :
– algèbre linéaire et matriciel
– polynômes et fonctions rationnelles
– interpolation et approximation
– optimisation linéaire, quadratique et non linéaire
– EDO / EDP
– Contrôle classique et robuste
– Traitement du signal
– Statistiques
– Graphisme 2D/3D
– ...
Le lancement de Scilab conduit à une fenêtre qui correspond (au gestionnaire de fenêtre près) à la figure 2.5.

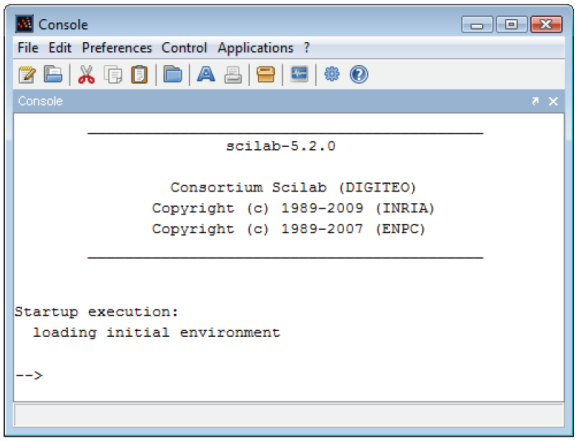


FIGURE 2.1 – Console Scilab obtenue au démarrage

2.1.1 Comment obtenir de l'aide ?

Une des manières les plus simples d'obtenir de l'aide avec Scilab est d'utiliser la fonction `help` intégrée à l'interface Scilab. Pour cela, il suffit de taper "`help`" dans la console et de valider (touche entrée).

Une fenêtre ***Help Browser*** s'ouvre et il est possible de naviguer dans le menu de gauche.

FIGURE 2.2 – Scilab Help Browser

Supposons que l'on souhaite obtenir de l'aide sur la fonction ***optim***, on peut la chercher dans la fenêtre ***Help Browser*** ou simplement en tapant dans la console ***help optim***.
On peut aussi accéder à une importante ressource documentaire sur la page web de Scilab :
`http://http://www.scilab.org/support/documentation`

Des mailing listes existent également.

Finalement, on pourra aussi accéder à des aides sous la forme de démonstrations dans le menu de la fenêtre console intitulé `?`.

2.1.2 Mode d'utilisation - console, éditeur, batch processing

Scilab s'utilise principalement par interaction avec le fenêtre console, au travers d'une boucle du type
lecture-évaluation-résultat
Voici une première interaction élémentaire :

```
--> s="Hello World!"
s=
Hello World!
--> disp(s)
Hello World!
```

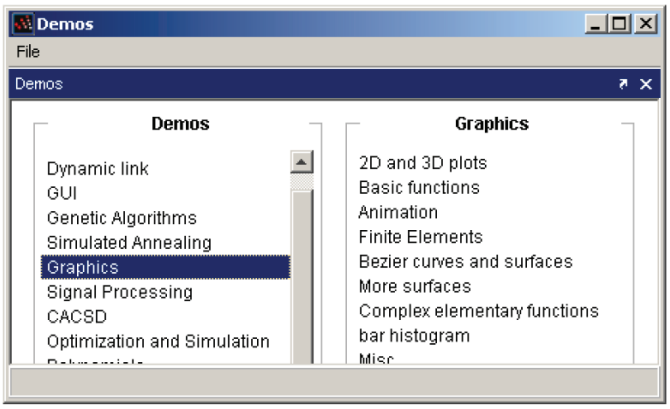


FIGURE 2.3 – Scilab Demos

Les flèches ↑ et ↓ permettent une navigation dans la liste des commandes précédemment saisies

La touche <TAB> lance un système de complétion facilitant la saisie.

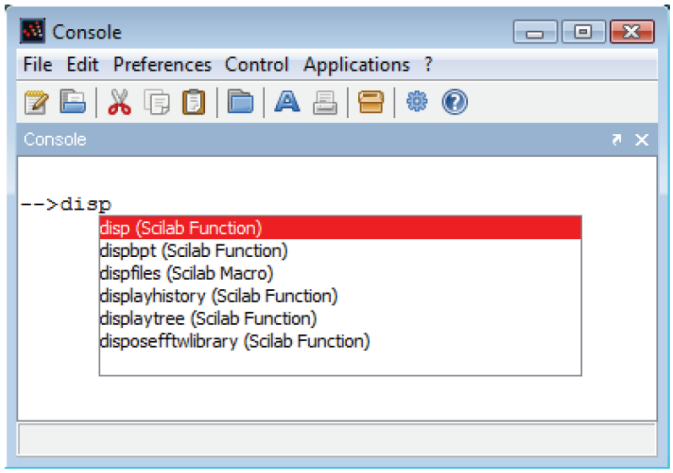


FIGURE 2.4 – Scilab - mécanisme de complétion à la saisie

L'environnement contient depuis la version 5.2, un **éditeur intégré**. On y accède en tapant directement à la console :

```
-> editor()
```

On peut aussi y accéder en ouvrant le menu `Application > Editor`.

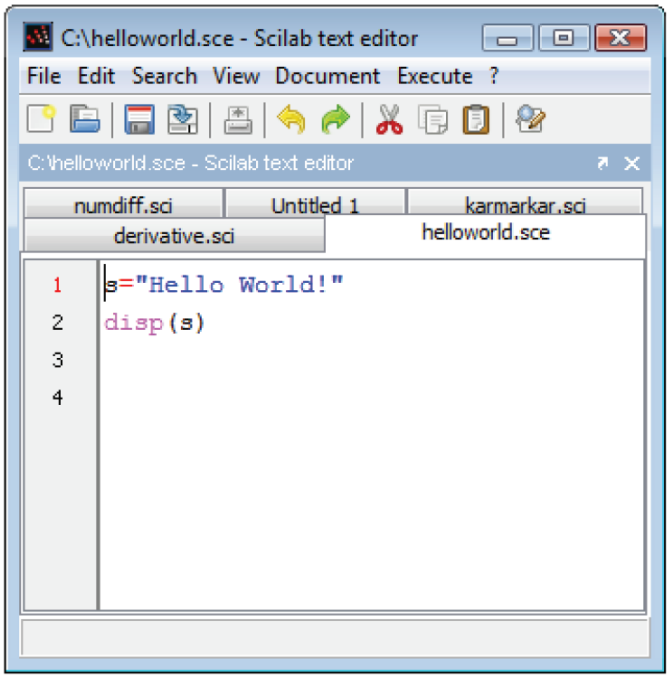


FIGURE 2.5 – Scilab - Editeur

On peut éditer plusieurs fichiers en simultané.

On s'intéressera au menu `Execute` ... voir la première session de TP pour l'expérimentation pratique.

Exécution de fichiers de scripts / commandes

A partir de l'éditeur intégré ou d'un autre éditeur, on peut construire un fichier de commandes qui sera introduit dans Scilab par la commande

```
exec nomfichier
```

- un ficheir d'extension `.sci` contient des descriptions de fonctions Scilab que l'on peut charger dans l'environnement Scilab sans être exécutée.
- un ficheir d'extension `.sce` contient des fonctions, mais aussi des commandes qui seront éxécutées au chargement.

Mode batch

On peut lancer Scilab en mode batch soit pour disposer d'un environnement simplifié (sans fenêtrage), soit pour lancer, en externe, des scripts et fichiers `.sce`

Suivant le système (Windows, Linux ou Mac), le nom de lancement différe :
– `Scilex` sous Windows
– `scilab` ou `scilab-cli` ou `scilab-adv-cli` sous Linux ou Mac

`Scilex -nw` et `scilab-adv-cli` lancent scilab en ligne de commande, sans fenêtrage.

`Scilex -f myscript.sce` lance scilab et le fichier de commande directement. Si ce ficheir se termine par `quit()`, scilab se referme alors directement après l'éxécution du fichier de commande.

2.2 Eléments de base

Scilab est un langage interprété permettant des constructions dynamiques de programmes. Généralement, comme pour *Matlab*, on peut dire que presque tout dans Scilab est une matrice (sauf des structures de données moins usuelles ; ce sera notamment des listes, par exemple, mais que l'on sera amené à peu utiliser dans la suite).

Malgré cette remarque, nous allons commencer par décrire des types de données simples (des matrices 1x1) avant de passer aux aspects plus généraux de traitements de matrice.

Variables réelles ou numériques

Voici une séquence d'instructions illustrant une manipulation très claasique des variables numériques. On notera l'opérateur d'affectation représenté par un seul signe = qu'il ne faut pas confondre avec le double symbole == qui désigne l'opérateur booléen de comparaison.

```
--> x=1
x=
1.
--> x=x*2
x=
2.
```

En ajoutant un point-virgule `;` à la fin d'une commande, on demande de ne pas générer d'affichage suite à l'exécution de la commande.
– x^2 ou $x**2$ désigne l'élévation au carré
– $x/y = xy^{-1}$
– $x\backslash y = x^{-1}y$

Noms de variables

Seuls les 24 premiers caractères d'un nom de variable sont significatifs. Un nom de variable est constitué :
– des lettres de l'alphabet en minuscules ou majuscules, avec différenciation de ces deux types
– des nombres de 0 à 9
– des caractères % _ # ! $?

Une variable commençant par % a une signification spéciale (vue plus tard).

Commentaires et lignes de continuation

Un commentaire est une ligne démarrant par `//`.

Une ligne se terminant par deux points successifs `..` sera prolongée par la ligne suivante.

```
--> // c'est un commentaire
--> x = 1 ..
--> + 2 ..
--> + 3
x=
6
```

Fonctions mathématiques élémentaires

Scilab intégre les fonctions mathématiques courantes :

acos	acosd	acosh	acoshm	acosm	acot	acotd	acoth
acsc	acscd	acsch	asec	asecd	asech	asin	asind
asinh	asinhm	asinm	atan	atand	atanh	atanhm	atanm
cos	cosd	cosh	coshm	cosm	cotd	cotg	coth
cothm	csc	cscd	csch	sec	secd	sech	sin
sinc	sind	sinh	sinhm	sinm	tan	tand	tanh
tanhm	tanm						

exp	expm	log	log10	log1p	log2	logm	max
maxi	min	mini	modulo	pmodulo	sign	signm	sqrt
sqrtm							

La plupart de ces fonctions prennent un argument et en retourne un autre. Elles sont "vectorisées", c'est à dire qu'elles manipulent des matrices.

Exemple :

```
--> x = cos(2)
x=
-0.4161468
--> y = sin (2)
y=
0.9092974
--> x^2 + y^2
ans=
1.
--> ans*2
ans=
2.
```

On notera la variable `ans` qui évalue l'expression courante et qui peut être réutilisée.

Variables mathématiques prédéfinies

Dans Scilab, plusieurs variables mathématiques sont prédéfinies ; elles commencent par % :
– % i : nombre complexe dont le carré vaut -1
– % e : constante d'Euler, base du mogarithme népérien
– % pi : constante π

Booléens

Les booléens sont des variables qui peuvent prendre deux valeurs :
– %t ou %T pour "true"
– %f ou %F pour "false'
'

Les opérateurs booléens sont listés ci-dessous

a&b	logical and
a|b	logical or
∼a	logical not
a==b	true if the two expressions are equal
a∼=b or a<>b	true if the two expressions are different
a<b	true if a is lower than b
a>b	true if a is greater than b
a<=b	true if a is lower or equal to b
a>=b	true if a is greater or equal to b

Nombres complexes

Les nombres complexes s'écrivent sous leur forme mathématique habituelle en décomposant la partie réelle et la partie imaginaire avec le nombre % i, imaginaire pur dont le carré vaut -1.

Listes des fonctions sur les nombre complexes

real	real part
imag	imaginary part
imult	multiplication by i, the imaginary unitary
isreal	returns true if the variable has no complex entry

Chaînes de caractères

Elles sont délimitées par des guillemets :

```
--> x = "coucou"
x=
coucou
```

La concaténation se fait par l'opérateur +.

Il existe de nombreuses fonctions de manipulation de chaînes de caractères que l'on peut découvrir et apprendre avec le systèmes d'aide, les démos et la documentation intégrée de scilab.

2.3 la manipulation de matrices et vecteurs

Dans cette section nous donnons les bases concernant la description des vecteurs et matrice puis l'apprentissage et la maîtrise du calcul matriciel se poursuit par l'intermédiaire d'exercices. L'objectif est d'obliger à la manipulation des aides et de la documentation intégrée de Scilab et de mieux comprendre les difficultés éventuelles du calcul matriciel par l'auto-apprentissage et les essais-erreurs-corrections.
Elémént de base en Scilab, une matrice peut contenr des nombres réels ou complexes. On peut la définir en respectant ls régles de constructions suivantes :
– Elle commence par un crochet ouvrant [et se termine par un crochet fermant]
– Les éléments d'une même ligne sont séparés par des blancs ou des virgules
– Les lignes dont séparées par des points vigules ;

```
--> A = [ 1 1 1; 2 4 8; 3 9 27]
--> y = [1 + %i, -2 - 3 * %i, -1]
```

Des matrices ou vecteurs spécifiques sont prédéfinis. Nous en donnons ici quelques uns :

– Construction d'une matrice identité
  `-> I = eye(4,4)`
– Construction d'une matrice diagonale
  `-> Y = diag(y)`
– Extraction de la diagonale d'une matrice
  `-> a = diag(A)`

Exercice 1 *Tester les instructions/constructions suivantes et comprendre leur fonctionnalité*
– `ones (3,4)`
– `zeros(3,4)`
– `triu(A)`
– `tril(A)`
– `rand (3,4)`

2.4 Visualiser un graphe simple

On va construire le graphe de la fonction

$$y = e^{-x}\sin(4x) \quad \text{pour } x \in [0, 2\pi]$$

On commence alors par créer un maillage de l'intervalle $[0, 2\pi]$ avec la fonction `linspace` :

```
-->x=linspace(0,2*%pi,101);
```

x est alors un vecteur de valeurs.

On calcule ensuite les valeurs de la fonction pour chaque composante du vecteur x, ce qui se fait en une seule instruction puisque Scilab manipule "naturellement" des vecteurs et matrices :

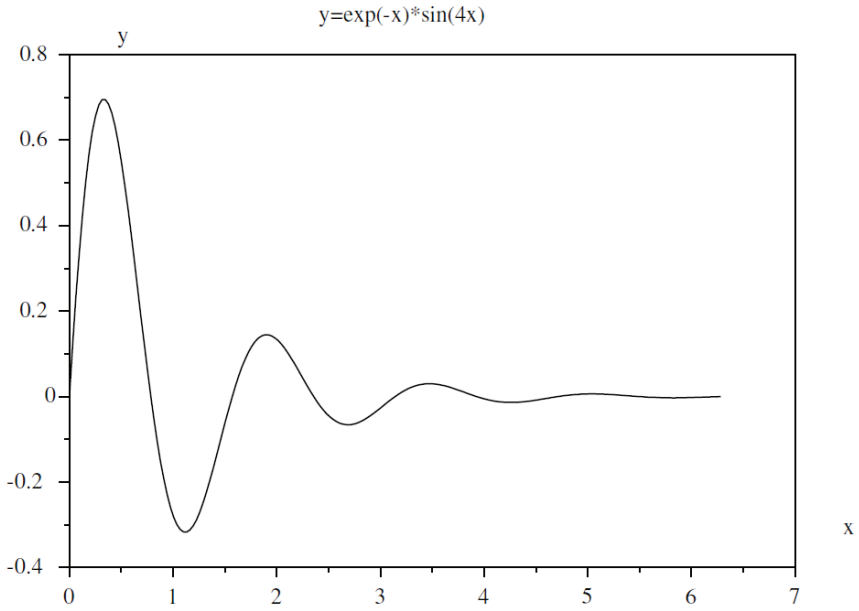
```
-->y=exp(-x).*sin(4*x);
```

Le tracé du graphique se fait alors par l'instruction `plot` ou `plot2d`. Elle est suivie ici par l'instruction `xtitle` qui permet d'afficher des légendes :

```
-->plot(x,y)
-->xtitle("y=exp(-x)*sin(4x)", "x", "y")
```

L'instruction permet de tracer une courbe passant par les points dont les coordonnées sont données dans les vecteurs x pour les abscisses et y pour les ordonnées. Les points sont reliés par des segments de droites.

Remarque : on étudiera plus tard des possibilités évoluées de tracé graphique.



2.5 Ecrire et exécuter un script

On peut écrire dans un fichier `nomfich` une suite de commandes et les faire exécuter par l'instruction :

```
-->exec('nomfich') // ou encore exec("nomfich")
```

On peut aussi charger le fichier par l'intermédiaire du menu "fichier", puis l'éxécuter par l'item "Exécuter".

Voici un petit exemple que l'on nomera "script.sce" :

```
// mon premier script Scilab
a = input(" Rentrer la valeur de a : ");
b = input(" Rentrer la valeur de b : ");
n = input(" Nb d'intervalles n : ");
// calcul des abscisses
x = linspace(a,b,n+1);
// calcul des ordonnees
y = exp(-x).*sin(4*x);
// representation graphique
plot(x,y)
xtitle("y=exp(-x)*sin(4x)","x","y")
```

2.6 Entrées-sorties et fichiers

2.6.1 Lecture par le clavier et affichage

Une saisie au clavier s'effectue comme cela a été illustré dans l'exemple de script donné dans la section précédente. On utilise la commande `input`.

Exemple 1 *Quelques saisies au clavier*

```
--> n=input("entrer la dimension n")
entrer la dimension n
--> 2
n =
  2.
--> a=input("entrer une matrice")
entrer une matrice
--> [1, 2; 3, 4]
a =
! 1.  2.!
! 3.  4.!
```

L'entrée d'une chaîne de caractères se fait encore avec la fonction `input`, en ajoutant un second paramètre qui est la constante "string".

Exemple 2 *Saisie d'une chaîne de caractères*

```
--> nomfichier = input("donner le nom du fichier", "string")
donner le nom du fichier
--> resultats.txt
nomfichier =
   resultats.txt
```

Une des commandes les plus usuels pour afficher les variables est `disp` qui contient autant d'arguments d'entrée que de variables à afficher.

Exemple 3 *Affichage de données avec "disp"*

```
--> a=[1, 2; 3, 4]; b="coucou"; c=2-sqrt(3);
--> disp(a,b,c)
 .2679492
 coucou
 ! 1.  2.!
 ! 3.  4.!
```

On remarquera l'ordre inversé d'affichage des données par rapport à l'ordre des arguments en entrée à la fonction "disp".

On peut aussi utiliser l'instruction "printf" qui est en fait celle qui est héritée du langage C.

Exemple 4 *Affichage avec la fonction "printf"*

```
--> a=64; b=sin(a);
--> printf('le sinus de %f \n est %f', a, b)
le sinus de 64.00000
est .920026
```

On ne donnera pas plus de détails ici sur les spécifications de `printf`. Le lecteur non habitué au langage C trouvera les informations dans tout manuel ou tutorial sur ce langage.

Exercice 2 *Utiliser l'aide pour savoir ce que fait la fonction* `x_message`. *Tester cette fonction en utilisant 1 ou 2 paramètres.*

2.6.2 Utilisation de fichiers

Deux fonctions de base servent à écrire des données dans un fichier.

`print (nomf, a)` ou `write(nomf, a)` : écrit la variable `a` dans le fichier dénommé `nomf`, en utilisant un format par défaut.

Exemple 5 *Ecriture de données non formatées dans un fichier*

```
--> a=[1,2,3; 4,5,6]; b(1)="Université"; b(2)="du Havre";
--> write("fic1",a); write("fic2",b);
```

Exercice 3 *Tester les instructions précédentes (et ensuite les suivantes) et éditer les deux fichiers générés pour observer leur contenu.*

On pourra relire ces fichiers par la commande `read`.

`a=read(nomf,m,n)` lits dans le fichier `nomf` les `m` premières lignes et les `n` premières colonnes et les affecte à la matrice `a`. Si on veut lire toutes les lignes du fichiers mais que l'on n'en connait pas le nombre, on remplace le second argument, noté `m` précédemment, par -1.

Exemple 6 *Lecture d'un fichier*

```
--> c=read(``fic1'',2,3);
--> d=read(``fic2'',-1,1,'(a)');
--> c, d
c =
! 1.  2.  3.!
! 4.  5.  6.!
d =
! Université !
!            !
! du Havre   !
```

Remarquons que nous avons ajouté un quatrième argument pour la lecture de "fic2". Cet argument désigne un format, ici '(a)' qui indique que le fichier contient des chaînes de caractères. Cette indication est ici indispensable sinon ce sont des valeurs numériques qui sont attendues à la lecture du fichier et une erreur est alors générée.

Plus généralement, les 2 instructions `write` et `read` peuvent avoir un argument supplémentaire correspondant à un format. Les formats d'écriture correspondent à ceux du langage C. Le lecteur est invité à consulter l'aide en ligne pour plus de précisions.

Exercice 4 *Rechercher dans l'aide en ligne, la fonction* `file` *qui permet de gérer des fichiers et tester quelques exemples de manipulation.*

2.7 La programmation en Scilab

Scilab propose un langage de programmation complet qui permet d'aller plus loin que d'utiliser des primitives toutes prêtes. Il s'agit d'un langage beaucoup plus "simple" que les langages classiques (C, C++, Java, ...) car il intégre déjà des primitives sophistiquées, notamment pour manipuler les matrices". Dans un soucis de simplification, le langage ne nécessite pas de déclarations qui sont gérées automatiquement par l'interpréteur. Une autre facilité appréciable est l'intégration d'une bibliothèque graphique.

L'inconvéniant majeur du langage Scilab par rapport aux langages classiques est qu'il n'est pas compilé et traduit directement en langage machine. Par ailleurs ses capacités de représentation de nombres cachent des routines soft de manipulation de grands nombre plutôt que des insctructions adhoc pour le processeur de la machine. Un programme Scilab nécessitera facilement 10 fois plus de temps qu'un programme compilé.

Scilab offre un interfaçage avec les langaes Fortran 77 et C, permettant de lancer des sous-programmes écrits et compilés dans l'un de ces langages.

2.7.1 Branchements et boucles

If - instructions conditionnelles

Exemple 7 *Test simple*

```
if (%t) then
   disp("Hello!")
end
```

Exemple 8 *Test avec alternative*

```
if (%f) then
   disp("Hello!")
else
   disp("Bye!")
end
```

On peut enchaîner plusieurs tests avec “elseif”.
L’expression de sélection qui se trouve entre paranthèses, après “if”, est une expression booléenne. On se souviendra que si c’est une comparaison, on utilisera le doule symbole “==” et non “=” qui est une affectation.

Select - alternatives

Exemple 9 *Instruction select*

```
i=2
select i
case 1 disp("One")
case 2 disp("Two")
case 3 disp("Three")
else disp("Beaucoup")
end
```

Seule l’instruction qui suit directement un case est exécutée (et pas les suivantes) si la comparaison est vraie. “Else” (qui est facultatif) sera effectuée si tout les tests précédents ont échoués.

For - boucle

Pour construire une boucle de type “for”, on utilise un index qui décrit un ensemble de valeurs. Ici, on va utiliser l’opérateur deux-points “ :” que l’on manipuler dans le chapitre précédent.

Exemple 10 *Séquence d’exemples d’utilisation de l’opérateur “ :”*

```
--> i=1:5
i =
  1. 2. 3. 4. 5.
--> i=1:2:5
i =
  1. 3. 5.
--> i=5:-1:1
i =
  5. 4. 3. 2. 1.
```

Exemple 11 *Construction de boucle "for" élémentaires*

```
for i=1:2:5
  disp(i)
end
```

En fait, l'indice de boucle peut parcourir tout type de séquences de données, comme des vecteurs, des lignes de matrices, des listes. Il faut que les éléments énumérés dans ces séquences soient de type entier, double, chaînes ou polynômes.

Exemple 12 *Construction d'une boucle "for" parcourant une matrice/vecteur*

```
v=[1.5 exp(1) %pi];
for x=v
  disp(x)
end
```

Remarque 1 *Un point important sur l'utilisation de la boucle "for" est de savoir si il n'est pas possible de la remplacer par une opération matricielle, c'est à dire une opération "vectorisée". En effet il peut y avoir un facteur de performance de l'ordre de 10 à 100 entre ces deux calculs au bénéfice des opérations vectorisées.*

While - boucle

Une boucle "while" conditionne la répétition d'une séquence d'instructions de manière conditionnée à la vérification d'une expression booléenne.

Exemple 13 *Une boucle "while" élémentaire calculant la somme des 10 premiers entiers*

```
s=0; i=1;
while (i<=10)
  s=s+i
  i=i+1
end
```

Pour ce calcul spécifique, on pourra utiliser plus efficacement la fonction prédéfinie "sum" :

Exemple 14 *Utilisation de la fonction "sum"*

```
--> sum(1:10)
ans =
   55.
```

Remarque 2 *La boucle "while" produit le même effet que la boucle "for" et il faut à nouveau se demander si une opération vectorielle ne peut pas la remplacer de manière plus efficace.*

Break et continue - sorties

Il s'agit de deux instructions qui sont à bannir pour les "puristes" de l'algorithmique (car traduisant une faiblesse d'analyse et des comportements invérifiables).

L'instruction "break" permet de sortir d'une boucle. Typiquement on vérifie une condition qui lorsqu'elle est satisfaite, ne nécessite plus de continuer la boucle (similaire à un traitement d'exception dans d'autres langages).

Exemple 15 *Utilisation basique de "break"*

```
s=0; i=1;
while (%t)
  if (i>10) then
    break
  end
  s=s+i
  i=i+1
end
```

L'instruction "continue" permet de signaler à l'intérieur d'une boucle que l'on doit "sauter" la fin des instructions restant à exécuter dans le corps de la boucle pour passer directement à l'itération suivante.

Exemple 16 *Utilisation basique de "continue"*

```
s=0; i=0;
while (i<=10)
  if (modulo(i,2)==0) then
    i=i+1
    continue
  end
  s=s+i
  i=i+1
end
```

Remarque 3 *La construction précédente, qui est un exemple de construction algorithmique à ne pas suivre, s'obtient plus simplement et efficacement avec la seule instruction :* `sum(1:2:10)`.

2.7.2 Les fonctions

La démarche essentielle qui permet d'élaborer des programmes en Scilab, est basée sur la décomposition fonctionnelle, c'est à dire à élaborer un programme de manière incrémentale en définissant des briques de base fonctionnelles. On va ainsi construire des fonctions de base que l'on va assembler ou utiliser dans d'autres fonctions

plus élaborées et abstraites.

Une fonction en Scilab s'utilise de la manière suivante :

```
outvar=myfunction(invar)
```

où
– `myfunction` est le nom de la fonction ;
– `invar` correspond aux arguments d'entrée qui doivent être fournis à la fonction pour qu'elle puisse s'exécuter ;
– `outvar` correspond aux arguments de sortie dont le but de l'exécution de la fonction est de les calculer.

Nous avons déjà utilisé beaucoup de fonctions qui sont prédéfinies dans Scilab, notamment les fonctions mathématiques usuelles telles que sin, cos, etc.

Les fonctions Scilab peuvent avoir un nombre arbitraire d'arguments d'entrée et d'arguments de sortie. On les distingue de la manière suivante :

```
[o1, o2, ..., on] = myfunction(i1, i2, ..., in)
```

Exemple 17 *Décomposition LU d'une matrice (exécuter les lignes d'instructions suivantes puis utiliser "help" pour comprendre ce qu'elles font)*

```
A=testmatrix("hilb",2)
[L,U]=lu(A)
[L,U,P]=lu(A)
```

Définir une fonction

Exemple 18 *Construction d'une fonction élémentaires*

```
function y = myfunction(x)
  y=2*x
endfunction
```

Ecrire la définition d'une fonction peut être long suivant la complexité de celle-ci. On peut la saisir en ligne dans l'environnement Scilab qui attendra de lire "endfunction" pour l'évaluer et renvoyer un prompt.

Si cette description est longue, il peut être préférable d'utiliser l'éditeur proposé dans Scilab et d'utiliser dans le menu de celui-ci, "Exécuter/Charger dans Scilab" pour la charger dans l'environnement interactif.

Une troisième alternative consiste à sauvegarder le code de la fonction (ou de plusieurs fonctions) dans un fichier d'extension ".sci"(fichier ne contenant que des définitions de fonctions) ou ".sce" (fichier contenant en plus des instructions exécutables), puis de d'utiliser la fonction "exec" dans l'environnement interactif avec une instruction qui pourrait être `exec("myFonctionsFile.sce")`

Exercice 5 *Définir la fonction précédente en utilisant successivement les 3 modes d'édition proposés.*

Remarque 4 *Il est essentiel dans le code de la fonction de trouver des instructions qui affectent des valeurs à chacun des arguments de sortie. Si ces affectations ne sont pas présentes, une erreur sera générée à l'appel de la fonction.*

Gestion des arguments de sortie

Voici un exemple de fonction renvoyant 2 arguments en sortie. Nous allons nous intéresser à différentes manière de récupérer ces arguments pour comprendre certaines spécificités de Scilab.

Exemple 19 *Fonction renvoyant deux arguments de sortie*

```
function [y1,y2]=simplef(x1,x2)
  y1=2*x1
  y2=2*x2
endfunction
```

Voici plusieurs résultats d'exécution :

```
--> [z1,z2] = simplef(1,2)
z2 =
   6.
z1 =
   2.
--> simplef(1,2)
ans =
   2.
--> z = simplef(1,2)
z =
   2.
```

La première instruction permet de récupérer les deux arguments de sortie comme décrit dans le corps de la fonction. La seconde et troisième instruction ne permettent de ne récupérer qu'une valeur qui est alors la valeur du premier argument de sortie (le second est alors perdu).

Nous donnons ci-dessous un nouvel exemple permettant d'illustrer la construction de fonctions récursives, ne nécessitant en Scilab aucun signalement spécifique.

Exemple 20 *Fonction récursive factorielle*

```
function f=fact(n)
  if n<=1 then f=1
  else f=n*fact(n-1)
  end
endfunction
```

Remarque 5 *Compléments vus en TP : des fonctions particulières de manipulation de fonctions permettant notamment de servir d'aide au débogage (disp, pause, resume, setbpt, delbpt, error, warning, type, typeof, argn, exists).*

Utiliser une fonction comme argument d'une autre fonction

Une fonction est elle-même une variable du type "function" et elle peut donc être passée comme argument d'une autre fonction.

Exemple 21 *Fonction possédant en argument d'entrée une autre fonction*

```
function y=f(x)
  y=sin(x) .* exp(-abs(x))
endfunction

function graphef(a,b,fct,n)
// n est un argument optionnel
// si il n'est pas présent, sa valeur par défaut est 61
  [lhs,rhs]=argn(0)
  if rhs==3 then n=61
  end
  x=linspace(a,b,n)
  y=fct(x)
  plot(x,y)
endfunction
```

On pourra alors tester la fonction `graphef` par les instructions suivantes :

```
--> graphef(-2*%pi,2*%pi,f)
--> graphef(-2*%pi,2*%pi,f,21)
```

Construire des fonctions de manière dynamique avec "deff", "evstr" et "execstr"

Scilab propose de définir des fonctions "in-line", grâce à une seule instruction grâce à la commande `deff` qui prend en argument d'entrée des chaînes de caractères où la fonction sera décrite.

Exemple 22 *Définition d'une fonction "in-line"*

```
--> deff('[s,d]=plusmoins(a,b)',['s=a+b', 'd=a-b'])
-->[p,q]=plusmoins(2,3)
 q =
  -1.
 p =
   5.
```

Cette possibilité est vraiment intéressante car elle permet de définir du code de manière dynamique : une chaîne de caractères contenant la totalité ou des éléments d'une fonction, peut-être lue au cours de l'exécution d'un programme et être utilisée en temps que fonction dans la suite du déroulement du programme.

Scilab offre aussi la possibilité d'évaluer une expression qui est décrite dans une chaîne de caractères, grâce à la fonction `evstr`. On peut également exécuter une instruction décrite dans une chaîne de caractères, grâce à la fonction `execstr`.

Exemple 23 *Conversion de chaîne de caractères en expression évaluable ou en instructions exécutables*

```
--> evstr("sqrt(3)/2")
ans =
   0.8660254
--> a=1; evstr("2 + a")
ans =
   3.
--> evstr(["a" "2"])
ans =
   ! 1.   2.!
--> execstr("A=rand(2,2)"); A
 A =
! 0.2113249  0.0002211 !
! 0.7560439  0.3303271 !
```

Bibliothèques de fonctions

(Baudin - 6.3)

2.8 Graphisme

(cf. chapitre 4 du document “Une introduction à Scilab” de Bruno Pinçon)

2.8.1 Compléments

les fonctions `fplot2d` et `fplot3d` permettent de faire le trace de courbes ou surfaces représentatives de fonctions mathématiques que l'on définit comme des fonctions Scilab.

Exemple 24 *Voici un exemple de tracé de fonction de la courbe représentative d'une fonction à une variable réelle, puis le tracé de la surface représentative d'une fonction à deux variables réelles.*

```
function y = f(x)
  y=sin(x)/x
endfunction

x=linspace(0.01, 4*%pi,101);
fplot2d(x,f)

function z = g(x,y)
  z=x*exp(-(x-1)^2-(y+0.25)^2)+y*exp(-x^2-y^2)+3
endfunction

x=linspace(-2, 3.5, 61);
y=linspace(-2.5, 2, 61);
xset("window",1);
fplot3d(x,y,g)
```

La surface représentative de la dernière construction est illustrée par la figure 2.6.

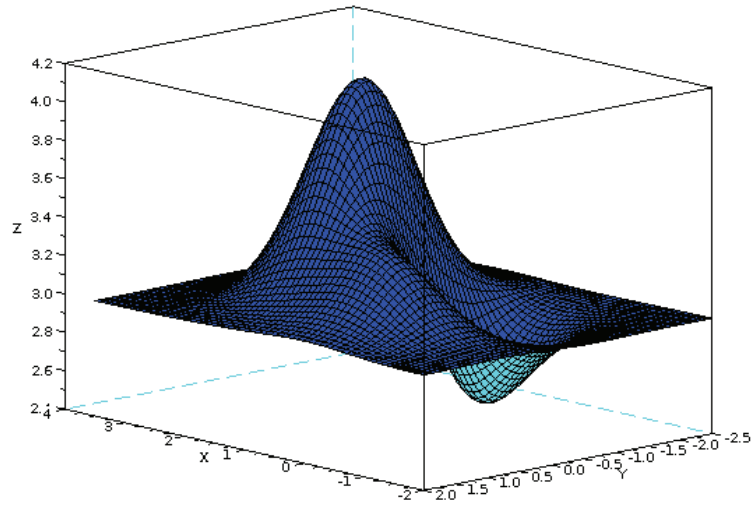


FIGURE 2.6 – Surface représentative d'une fonction avec `fplot3d`

2.9 Modélisation et systèmes différentiels

Scilab permet de résoudre *numériquement* des équations différentielles. On ne cherche donc pas ici une solution analytique de la solution d'une équation différentielle mais on calcule une approximation de la solution à une succession de pas de temps, à partir d'une condition initiale.

Soit une équation avec une condition initiale (problème de Cauchy) :

$$\left\{ \begin{array}{rcl} \dfrac{du(t)}{dt} & = & f(t, u(t)) \\ u(t_0) & = & u_0 \end{array} \right. \tag{2.1}$$

où $u : I\!R \to I\!R^n$, $u_0 \in I\!R^n$ et $f : I\!R \times I\!R^n \to I\!R^n$.
Le théorème de Cauchy-Lipschitz nous dit que si f est lipschitzienne par rapport à sa deuxième variable, sur un intervalle fermé contenant u_0 [1], alors il existe une solution unique pour la fonction u, donnant ainsi une valeur unique de cette fonction en une valeur t donnée, à condition de ne pas sortir de l'intervalle dans lequel la condition de Cauchy-Lipschitz est énoncée. Dans la suite on supposera que cette condition de régularité de la fonction f est vérifiée.

1. c'est à dire que pour cet intervalle fermé I, il existe k réél strictement positif tel que pour toute valeur de t et tout couple de valeurs de (u, v) dans l'intervalle I, $\|f(t, u) - f(t, v)\| < k\|u - v\|$

La fonction `ode` est la fonction Scilab qui va permettre de calculer une solution numérique, c'est à dire de l'"intégrer" sur un intervalle démarrant en t_0, en partant du vecteur colonne u_0.

Pour cela, il faut commencer par définir la fonction f du problème de Cauchy décrit ci-dessus (2.1). On la définit comme une fonction Scilab classique :

```
function [f] = SecondMembreEDO(t,u)
  // on décrit ici les composantes de f
endfunction
```

Remarque 6 *Même si la fonction f est autonome (c'est à dire indépendante de t), il faudra quand même mettre t comme premier argument de ce second membre, afin qu'ultérieurement, il soit identifié comme la variable de la solution cherchée.*

Exemple 25 *On considère l'équation de Van der Pol :*

$$y'' = c(1 - y^2)y' - y$$

que l'on reformule classiquement comme un système de deux équations différentielles du premier ordre, en posant

$$\begin{array}{rcl} u_1(t) & = & y(t) \\ u_2(t) & = & y'(t) \end{array}$$

Soit le système équivalent

$$\frac{d}{dt}\begin{bmatrix} u_1(t) \\ u_2(t) \end{bmatrix} = \begin{bmatrix} u_2(t) \\ c(1 - u_1^2(t))u_2(t) - u_1(t) \end{bmatrix}$$

Ce que l'on décrit par la fonction Scilab suivante, en fixant $c = 0.4$:

```
function [f] = vanDerPol(t,u)
  f(1)=u(2)
  f(2)=0.4*(1-u(1)^2)*u(2) - u(1)
endfunction
```

Pour résoudre numériquement l'équation, il faut donc définir une distribution uniforme de valeurs de t sur un intervalle, à partir de t_0, avec par exemple, la fonction $linspace$. On appelle ensuite la fonction `ode` *dont les arguments sont les suivants :*
- *les deux premiers paramètres correspondent à la condition initiale du problème de Cauchy, ce qui correspond ici au vecteur $[u_1(t_0), u_2(t_0)]$ et t_0 ;*
- *le troisième est le vecteur des valeurs successives de t ;*
- *le quatrième est la fonction f du problème de Cauchy (2.1).*

Ceci correspond ainsi à deux lignes de code Scilab prenant la forme suivante :

```
t = linspace(t0,T,m);
[U] = ode(u0, t0, t, vanDerPol)
```

On récupère alors une matrice U telle que $U(i,j)$ est la solution approchée de $u_i(t(j))$.

On donne ci-après un traitement complet de l'équation de Van Der Pol. On commence par tracer le champ de vecteur correspondant de la fonction f du problème de Cauchy pour une valeur quelquonque de t, ici prise à 0, sachant que cette équation est autonome, ce champ de vecteur sera le même quelque soit la valeur de t. Selon la construction du problème de Cauchy (2.1), ce champ vectoriel va décrire les tangentes aux trajectoires correspondant à des portraits de phase, c'est à dire aux courbes $(u_1(t), u_2(t))$, paramétrées par le temps. On effectue ensuite une résolution numérique de l'équation différentielle (avec `ode`*).On trace alors par-dessus le champ vectoriel, la courbe de portrait de phase démarrant à la condition initiale. On trace ensuite dans deux autres fenêtres graphiques, l'évolution de chacune des composantes de la solution par rapport au temps.*

```
function [f] = vanderpol(t,u)
  f(1)=u(2)
  f(2)=0.4*(1-u(1)^2)*u(2)-u(1)
endfunction

n=30;
delta=5;
x=linspace(-delta, delta,n); // on prend y=x
clf()
fchamp(vanderpol,0,x,x)
//tracé du champs de vecteur à l'instant t=0

m=500; T=30;
t=linspace(0,T,m);
u0 = [-2.5; 2.5];
plot2d(u0(1),u0(2),style=-9)
          //marquage de la condition initiale
[u] = ode(u0,0,t,vanderpol);
plot2d(u(1,:)', u(2,:)', style=2)
xtitle("Equation de Van Der Pol :
          plan de phase (y(t), dy(t)/dt)", "y", "dy/dt")

xset("window",1);
plot2d(t', u(1,:)')
xtitle("Equation de Van Der Pol : y(t)", "t", "y")

xset("window",2);
plot2d(t', u(2,:)')
xtitle("Equation de Van Der Pol : dy(t)/dt",
```

```
          "t", "dy/dt")
```

Remarque 7 *Derrière la résolution numérique de la fonction* `ode`*, il y a plusieurs algorithmes possibles que l'on peut contrôler soi-même en indiquant celui que l'on souhaite utiliser (non décrit ici ... voir l'aide si nécessaire). Par défaut, un processus adaptatif est utilisé : il est capable de passer d'une méthode explicite standard (ici méthode d'Adams prédicteur/correcteur) à une méthode spécifique à une équation "raide" si l'équation est diagnostiquée en tant que telle, c'est à dire qu'elle s'intégre difficilement avec une méthode explicite (une méthode de Gear est alors utilisée).*