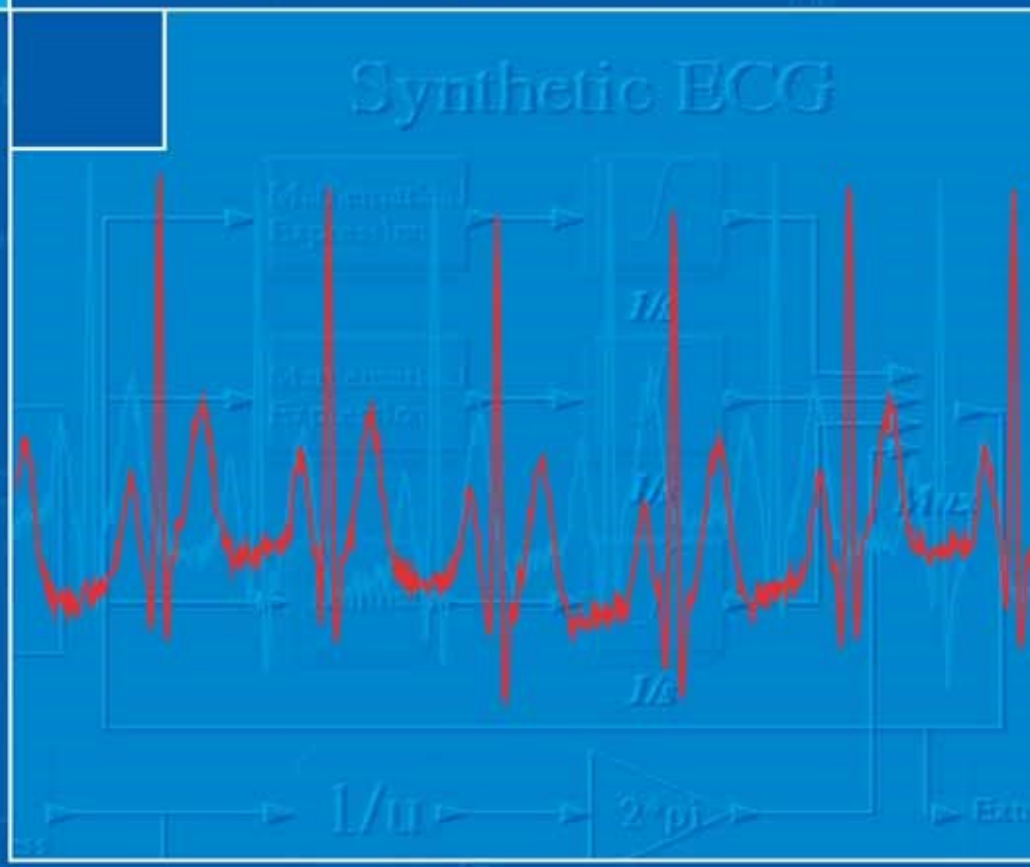
J.-P. Chancelier, F. Delebecque,
C. Gomez, M. Goursat,
R.Nikoukhah, S. Steer

Collection IRIS
dirigée par Nicolas Puech



Introduction à SCILAB

Deuxième édition

Springer
SCILAB
INRIA

Introduction à Scilab

Deuxième édition

Springer
Paris
Berlin
Heidelberg
New York
Hong Kong
Londres
Milan
Tokyo

J.-P. Chancelier, F. Delebecque,
C. Gomez, M. Goursat,
R. Nikoukhah, S. Steer

Introduction à Scilab
Deuxième édition

Springer

Jean-Philippe Chancelier
CERMICS
École nationale des Ponts et Chaussées
Cité Descartes
77455 Marne-la-Vallée Cedex 02

François Delebecque
INRIA Rocquencourt
Domaine de Voluceau-Rocquencourt
BP 105
78153 Le Chesnay

Claude Gomez
INRIA Rocquencourt
Domaine de Voluceau-Rocquencourt
BP 105
78153 Le Chesnay

Maurice Goursat
INRIA Rocquencourt
Domaine de Voluceau-Rocquencourt
BP 105
78153 Le Chesnay

Ramine Nikoukhah
INRIA Rocquencourt
Domaine de Voluceau-Rocquencourt
BP 105
78153 Le Chesnay

Serge Steer
INRIA Rocquencourt
Domaine de Voluceau-Rocquencourt
BP 105
78153 Le Chesnay

ISBN 13 : 978-2-287-25247-1 Springer Paris Berlin Heidelberg New York

Printed in France
Springer-Verlag France est membre du groupe Springer Science + Business Media

Maquette de couverture : Jean-François MONTMARCHÉ

Collection *IRIS*
Dirigée par Nicolas Puech

Ouvrages parus :

– *Méthodes numériques pour le calcul scientifique. Programmes en Matlab*
A. Quarteroni, R. Sacco, F. Saleri, Springer-Verlag France, 2000

– *Calcul formel avec MuPAD*
F. Maltey, Springer-Verlag France, 2002

– *Architecture et micro-architecture des processeurs*
B. Goossens, Springer-Verlag France, 2002

– *Introduction aux mathématiques discrètes*
J. Matousek, J. Nesetril, Springer-Verlag France, 2004

– *Les virus informatiques : théorie, pratique et applications*
É. Filiol, Springer-Verlag France, 2004

– *Introduction pratique aux bases de données relationnelles. Deuxième édition*
A. Meier, Springer-Verlag France, 2006

– *Bio-informatique moléculaire. Une approche algorithmique*
P.A. Pevzner, Springer-Verlag France, 2006

– *Algorithmes d'approximation*
V. Vazirani, Springer-Verlag France, 2006

– *Techniques virales avancées*
É. Filiol, Springer-Verlag France, 2007

– *Codes et turbocodes*
C. Berrou, Springer-Verlag France, 2007

À paraître :

– *Forces de la programmation orientée objet. De la théorie à la pratique*
J. Pasquier, P. Fuhrer, A. Gachet, Springer-Verlag France, 2008

Avant-propos

Dans le monde scientifique, l'utilisation des logiciels de calcul numérique est aujourd'hui indispensable. Des chimistes qui simulent leurs modèles de processus aux ingénieurs qui conçoivent des systèmes de régulation, des spécialistes des télécommunications qui mettent au point des algorithmes de traitement du signal aux spécialistes de finance qui manipulent des modèles statistiques, tous ont des besoins importants en calcul numérique. Scilab, qui est largement diffusé dans le monde universitaire et industriel depuis quelques années, est un logiciel libre de calcul numérique.

Pendant de nombreuses années, Scilab a été essentiellement développé par des chercheurs de l'INRIA et de l'ENPC avec de nombreuses contributions extérieures, souvent sous forme de boîtes à outils. Scilab est maintenant pris en charge par un consortium auquel participent financièrement une vingtaine de grandes sociétés (voir `www.scilab.org` où l'organisation est décrite).

Scilab n'est pas qu'une super-calculette : c'est un puissant environnement logiciel pour le développement d'applications scientifiques. Ces dernières nécessitent le plus souvent la manipulation de matrices et vecteurs et c'est pour cela que Scilab, comme d'autres logiciels tels que Matlab ou Octave, utilise une syntaxe adaptée à la manipulation de ces objets. En outre, Scilab propose des centaines de fonctions de calcul numérique regroupées en bibliothèques qui couvrent des domaines comme la simulation, l'optimisation, l'automatique ou le traitement du signal, ce qui facilite le développement d'applications particulières.

Bien que la plupart des fonctions soient de type mathématique, on peut aussi bien utiliser Scilab comme un langage de programmation généraliste. Il est par exemple tout à fait possible de développer un tableur dans Scilab.

Écrit par les développeurs de Scilab, l'ouvrage devrait permettre au lecteur de maîtriser le logiciel et de développer ses propres applications. Destiné à un large public d'étudiants, chercheurs, enseignants et ingénieurs, il propose un apprentissage du logiciel à partir d'exemples.

Cette deuxième édition a été l'occasion d'en actualiser le contenu pour prendre en compte les évolutions du logiciel entre la version 2.6 et l'actuelle

version 4.1. Ces évolutions concernent principalement les fonctionnalités graphiques, l'amélioration de l'intégration dans l'environnement Windows et le développement de Scicos, boîte à outils Scilab pour la modélisation et de simulation

Les sources des exemples du livre peuvent être téléchargées sur le site web de Scilab : `www.scilab.org/contrib/index_contrib.php`. L'archive à télécharger contient 4 répertoires : `scilab` pour les scripts Scilab, `scicos_diag` pour les schémas Scicos, `C` pour les codes en langage C et `help` pour les exemples relatifs à l'aide en ligne. Les chemins relatifs des fichiers sont généralement indiqués dans le texte par une référence de la forme : Fichier source : *scilab/mafonc3.sci*. Ce livre et ses exemples concernent la version 4.1 de Scilab mais les exemples devraient être utilisables avec toutes les versions postérieures à la version 4.0.

Le livre est divisé en deux parties. La première concerne le langage de programmation et la deuxième présente quelques domaines d'application. Dans la première partie le lecteur apprendra à utiliser le logiciel en mode interactif comme une calculatrice, puis il apprendra à définir ses propres fonctions et à utiliser les nombreuses facilités disponibles dans l'environnement Scilab comme la gestion des entrées-sorties, le graphique, le développement d'interfaces homme-machine, l'interfaçage avec des programmes extérieurs en C, C++ ou FORTRAN, et le débogage.

Dans la deuxième partie, on introduit quelques fonctionnalités qui sont essentielles dans la plupart des applications scientifiques telles que la résolution d'équations, l'optimisation, la simulation ou des outils statistiques. Sans être bien sûr exhaustif, on a surtout cherché à donner à travers des exemples simples, une idée des types d'applications qui peuvent être développées avec Scilab. On présente aussi dans cette partie Scicos, une boîte à outils dédiée à la modélisation par schéma-blocs et la simulation des systèmes dynamiques. Cette méthode de modélisation est de plus en plus utilisée dans le monde industriel, car elle permet de développer du code modulaire et réutilisable. Le lecteur aura les éléments de base pour construire des schémas de simulation à partir des blocs prédéfinis et disponibles dans des palettes mais aussi pour construire de nouveaux blocs.

Les auteurs remercient Bruno Petazzoni et Nicolas Puech pour leur aide dans la mise au point de cet ouvrage.

Sommaire

Première partie

Le logiciel Scilab

Chapitre 1

Vue d'ensemble

1.1 Qu'est-ce que Scilab ?

Il existe deux types de logiciels de calcul scientifique : les logiciels de calcul symbolique qui schématiquement « font des mathématiques » et les logiciels de calcul numérique, qui sont plutôt conçus pour les applications des mathématiques. Dans la première catégorie on trouve, entre autres, Maple, Mathematica et MuPad. Ces logiciels sont utilisés depuis quelques années dans les classes préparatoires scientifiques.

Dans la deuxième catégorie, où le marché est plus large, on trouve essentiellement les logiciels commerciaux Matlab et Xmath. S'y ajoute Scilab, qui est un logiciel libre[1], distribué avec son code source.[2]

Scilab est un gros logiciel qui comporte plusieurs dizaines de milliers de fichiers pour plus de 1 200 000 lignes de code : en langage FORTRAN et C pour les fonctions de base, en langage Scilab pour les librairies spécialisées et en anglais et français pour l'aide en ligne (`help`) et les documentations. La gestion représente à elle seule plus de 20000 lignes de code (fichiers de configuration, scripts ...).

Scilab comporte un interpréteur, des objets et des fonctions bien adaptés au calcul numérique et à la visualisation des données. En plus des vecteurs et des matrices (qui peuvent contenir des nombres réels ou complexes, des entiers, des chaînes de caractères, des polynômes, ...), on peut définir dans Scilab des objets plus complexes à partir de structures et surcharger les opérations correspondantes. De plus, l'utilisateur peut rajouter à Scilab des fonctions écrites en langage C, C++ ou FORTRAN en les connectant dynamiquement à Scilab.

Scilab contient aussi de nombreux outils de visualisation graphique : des graphiques 2D et 3D, des tracés de contours, des tracés paramétriques, des

[1] On peut aussi citer les logiciels libres Octave et FreeMath.

[2] Bien que son utilisation soit entièrement libre, Scilab est un logiciel protégé (voir le fichier `licence.txt`).

animations, etc. Les graphiques peuvent être exportés aux formats Xfig, Postscript, Gif, Latex, PPM ainsi que EMF (format vectoriel spécifique à Windows). En plus des boîtes de dialogues pré-définies, l'interface avec Tcl-Tk permet de construire des boîtes de dialogues complexes.

Sur le site `www.scilab.org`, Scilab est distribué sous forme binaire pour les PC-Linux, Windows 9X/2000/XP et les stations Unix Solaris, mais construire Scilab à partir de la version source ne pose aucun problème sur les stations mentionnées (par exemple si on souhaite utiliser la version développement pour une nouvelle fonctionnalité ou une correction de bug). Le site scilab.org renvoie à d'autres sites sur lesquels on peut télécharger la version Mac OS X. Il est enfin possible par le newsgroup `comp.soft-sys.math.scilab` de demander si des portages plus spécifiques existent.

1.2 Comment démarrer ?

1.2.1 Installation du logiciel

L'installation dépend du système d'exploitation (Windows, Unix, Linux). On peut installer une version binaire ou compiler Scilab à partir du code source. Cette dernière opération nécessite des compilateurs C et FORTRAN (ou `f2c`). Pour la version Windows, deux voies sont possibles : utiliser Cygwin si on veut rester dans l'univers du logiciel libre ou utiliser les outils Microsoft : Visual C++ ou Visual Express qui est disponible gratuitement. La compilation sous Windows est parfois plus délicate que sous Linux. Toutes les distributions Linux contiennent les compilateurs C et FORTRAN. Sur les stations de travail Unix, si les compilateurs natifs ne sont pas disponibles, on peut utiliser les compilateurs GNU.

Dans ce livre, le mot-clé `SCI` désigne le répertoire où Scilab est installé. De plus, nous utiliserons la syntaxe du monde UNIX pour désigner les répertoires. Donc le chemin `SCI/routines/machine.h` représente le fichier `machine.h` dans le sous-répertoire `routines`. Sous Windows, il faut remplace les `/` par des `\` .

1.2.2 Démarrage

En lançant Scilab, on obtient la fenêtre de la figure 1.1 sous Unix/Linux ou 1.2 sous Windows. Il est à noter que l'aspect, voire l'organisation des menus peuvent être différents. De plus sur la version Windows uniquement, si l'utilisateur a choisi une installation en Français les textes des menus sont en français.

À l'invite (le « prompt »), représenté graphiquement par le symbole `-->`, on passe une commande, Scilab l'interprète, l'évalue, donne le résultat puis redonne une invite.

Pour explorer Scilab, on peut commencer par examiner les exemples, ce qui se fait, sous Windows, en cliquant sur le bouton ? puis `Démonstrations Scilab`

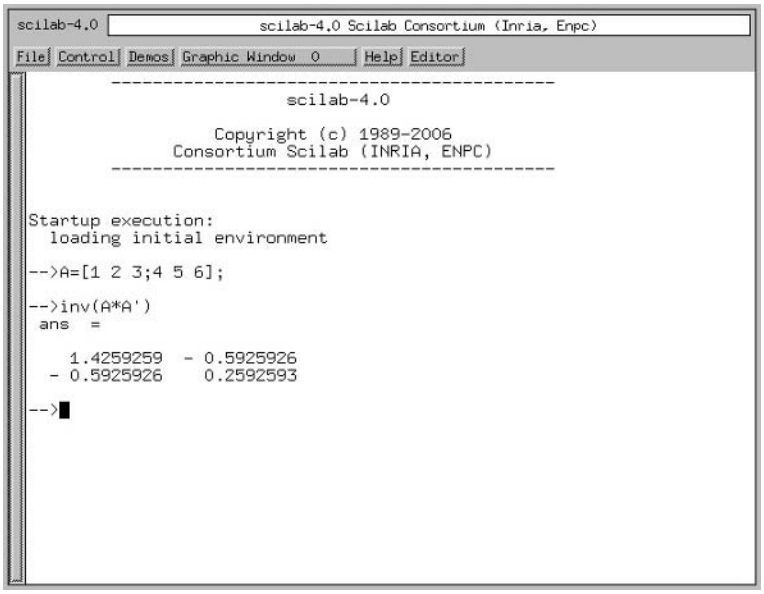


Figure 1.1 – La fenêtre principale Scilab sous Unix.

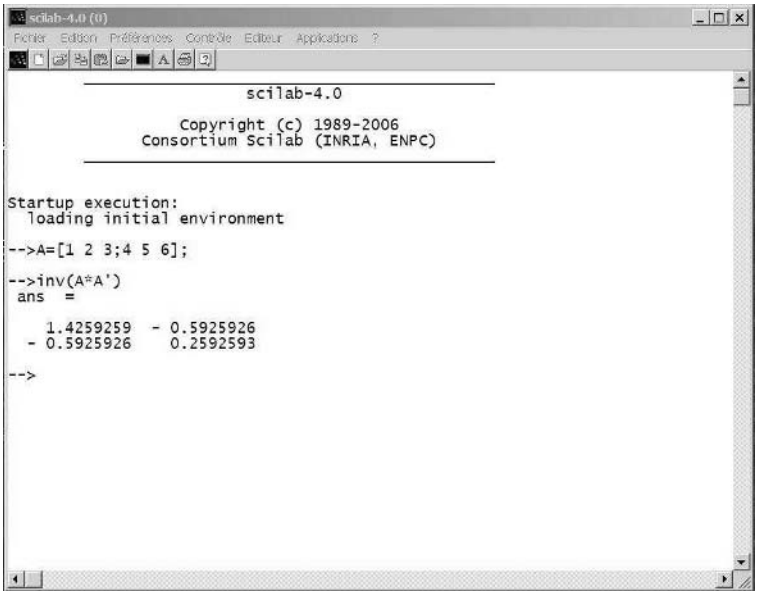


Figure 1.2 – La fenêtre principale Scilab sous Windows.

ou directement sur le menu `Demos` sous Unix. En observant les exemples, on commence à voir les capacités du logiciel et quelques unes de ses boîtes à outils spécialisées.

Pour chaque démonstration, l'utilisateur peut voir le code source : les objets manipulés sont surtout des vecteurs et des matrices numériques, ce qui donne un code très compact et lisible avec des notations matricielles naturelles. Le gain par rapport à un codage en C (ou FORTRAN) est énorme. En outre, il n'y a pas de déclaration de types à faire, pas de compilation à exécuter, ni d'allocation de mémoire à gérer, tout est automatique. Le prix à payer en temps est, bien sûr, celui de l'interprétation qui peut devenir conséquent pour certaines applications.

1.2.3 Éditeur de commandes

La fenêtre de commande de Scilab permet l'édition de l'instruction courante et le rappel des instructions précédemment entrées. Le positionnement du curseur peut s'effectuer en utilisant les flèches directionnelles ($\leftarrow\!\!\uparrow\!\!\downarrow\!\!\rightarrow$) du clavier ou en utilisant des caractères de contrôle « à la Emacs ». La notation Ctrl-<touche> indique que l'on presse la touche <touche> en maintenant la touche Ctrl enfoncée. Ainsi Ctrl-b décale le curseur d'un caractère vers la gauche, Ctrl-f d'un caractère vers la droite, Ctrl-a positionne le curseur en début de ligne, Ctrl-e positionne le curseur en fin de ligne. Ctrl-k efface tous les caractères de la ligne qui suivent le point d'insertion, Ctrl-y colle les caractères précédemment coupés à partir du point d'insertion (sous Unix seulement). Les contrôles Ctrl-p et Ctrl-n permettent respectivement d'accéder aux lignes précédente et suivante.

Il est aussi possible de rappeler une instruction précédente en entrant la séquence suivante !<début> où <début> représente les premiers caractères de la ligne recherchée.

Toutes les commandes saisies sont automatiquement sauvées, lorsque l'on quitte Scilab, dans le fichier nommé `.history.scilab` dans le répertoire donné par la variable Scilab `SCIHOME`. Bien évidemment, ces commandes sont disponibles, après avoir quitté Scilab, lors de l'ouverture d'une nouvelle session.

1.2.4 Documentation

Il existe une documentation en ligne qui contient les descriptions détaillées de toutes les fonctions disponibles, accompagnées d'exemples typiques d'utilisation. Les exemples peuvent être exécutés par un copier-coller de la fenêtre help dans la fenêtre Scilab. Cette documentation est accessible à travers les commandes `help` et `apropos`. Si le nom exact de la fonction est connu, on utilise la commande `help` suivie par le nom de la fonction ; une fenêtre spéciale s'affiche alors (figure 1.3), sinon on utilise la commande `apropos` suivie par un mot clé.

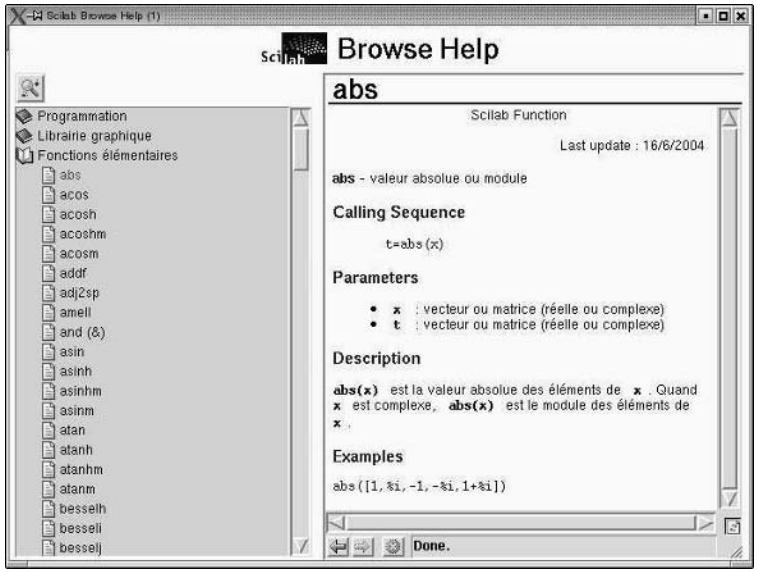


Figure 1.3 – La fenêtre d'aide.

Au téléchargement de Scilab on peut choisir la version anglaise ou la version française : en fait cette dernière n'est que partiellement en français ; elle a été définie pour satisfaire les besoins usuels d'un étudiant de niveau licence.

Toute cette documentation est aussi disponible en utilisant les menus de la fenêtre principale : sous Windows en cliquant sur le menu ? de la fenêtre principale puis sur le sous-menu `Aide Scilab`, sous Unix sur le menu `Help` puis sur le sous-menu `Help Browser`. On obtient une fenêtre de navigation (figure 1.4) présentant une liste de toutes les fonctions classées par thème. La sélection d'un thème montre l'ensemble des fonctions relatives à ce thème, la sélection d'un nom ouvre alors la page d'aide correspondante. La sélection de l'icône  ouvre un champs de saisie permettant de spécifier le nom que l'on recherche, la sélection de l'icône  retourne à la présentation des thèmes.

À la documentation en ligne, il faut ajouter un certain nombre de documents disponibles sur le site de Scilab : la plupart concernent une version antérieure de Scilab mais leur utilité est cependant très appréciable.

D'autres livres ([4], [6], [2], [1], [21], [7], [14], [5], [19]) fournissent aussi une documentation appréciable sur Scilab en général ou sur certaines de ses boîtes à outils. Les démonstrations sont souvent des exemples intéressants de programmation. Par exemple, les démonstrations graphiques permettent à l'utilisateur de comprendre l'essentiel des fonctionnalités graphiques. De manière générale,

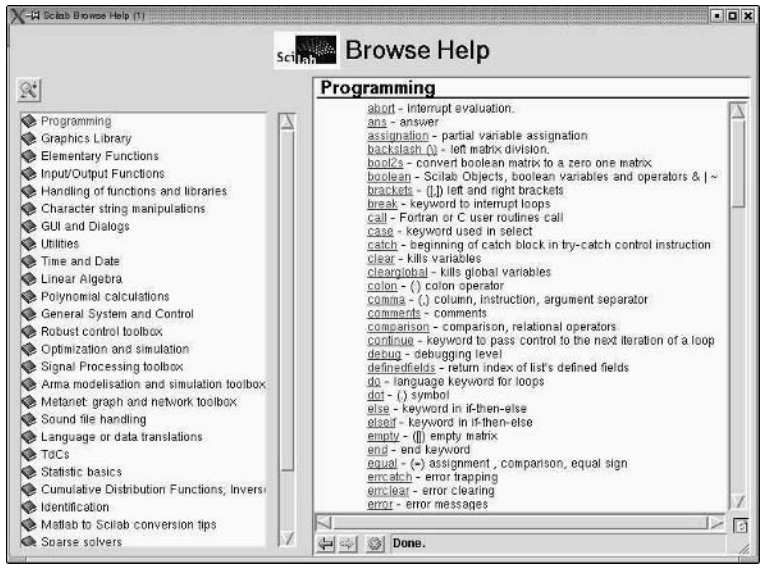


Figure 1.4 – La fenêtre de navigation de l'aide.

ces démonstrations peuvent être utilisées comme point de départ pour des applications plus complexes.

Les démonstrations donnent des exemples de graphiques, de traitement du signal, de commande, et de simulation de systèmes dont un modèle complet du vélo, de schémas Scicos et beaucoup plus.

1.3 Éditeur de texte intégré

L'utilisateur de Scilab passe l'essentiel de son temps à définir et à mettre au point ses fonctions en faisant de nombreux allers et retours entre son éditeur et Scilab. Un programme Scilab se réduit souvent à quelques lignes car les primitives de calcul et de manipulation d'objets fournies sont nombreuses et puissantes. Ces programmes sont définis (paragraphe 3.3) dans des fichiers que l'utilisateur construit à l'aide de son éditeur préféré, puis transmis à Scilab.

L'environnement Scilab contient un éditeur de texte `scipad` intégré et adapté à Scilab. Cet éditeur peut être lancé par le menu `Editeur` de la fenêtre principale de Scilab, ou par l'instruction `scipad()`. L'éditeur comporte toutes les fonctionnalités qu'un programmeur chevronné peut souhaiter : complétion, la mise en correspondance des parenthèses, recherche de mots dans le programme, colorisations diverses, manipulations de programmes, insertion de points d'arrêt, visualisation de variables... La figure 1.5 montre la fenêtre de l'éditeur avec un exemple très simple d'utilisation de la touche `complétion` (choix laissé à l'utili-

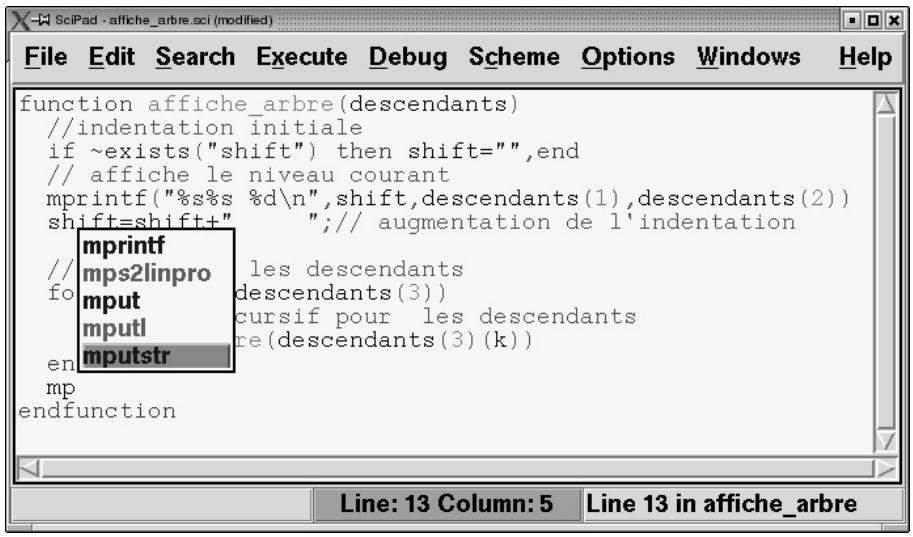


Figure 1.5 – La fenêtre de l'éditeur associé et mode complétion.

sateur parmi plusieurs possibilités) : après avoir tapé `mp` la touche `complétion` ouvre une petite sous-fenêtre avec toutes les commandes Scilab commençant par `sq`. La figure 1.6 présente les outils d'aide à la mise au point. L'utilisateur peut aussi utiliser son éditeur de texte préféré (sous Unix, il s'agit en général de `Vi` ou d'`Emacs` pour lequel existe d'ailleurs un mode Scilab ; sous Windows cela peut être `Notepad` ou `UltraEdit`).

1.4 Scilab sur la toile

La version la plus récente du logiciel Scilab, sa documentation et un grand nombre de contributions extérieures (boîtes à outils) se trouvent sur le site Web de Scilab :

`http://www.scilab.org`

Il existe aussi un newsgroup international dédié à Scilab :

`comp.soft-sys.math.scilab`

Enfin l'équipe de Scilab s'efforce dans la mesure du possible de répondre aux mails envoyés à `scilab@inria.fr`.

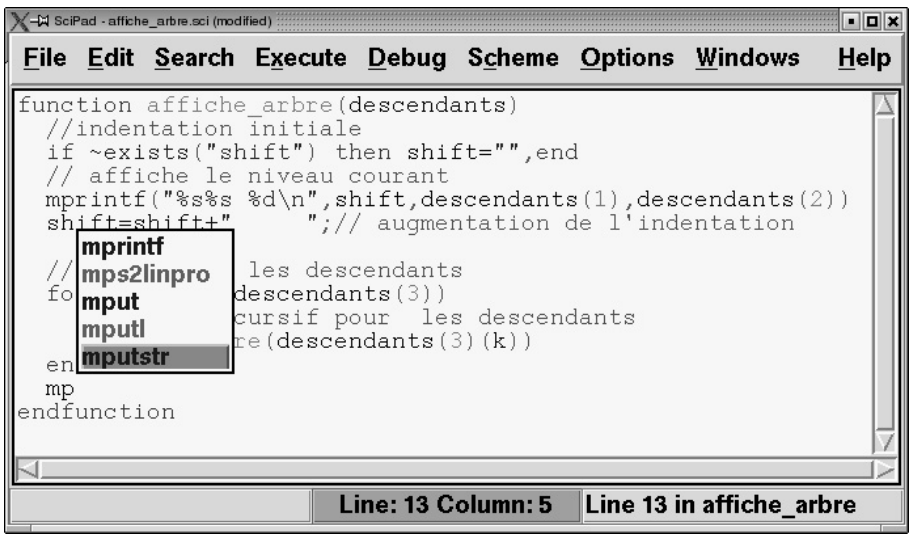


Figure 1.6 – La fenêtre de l'éditeur associé et outils de mise au point.

Chapitre 2

Définition et manipulation des objets

Le langage Scilab contient des opérateurs puissants permettant la manipulation des objets de base, lesquels sont nombreux et variés.

2.1 Objets Scilab

Scilab propose une importante collection d'objets de base. Les plus élémentaires d'entre eux sont les nombres flottants et les chaînes de caractères (« strings »). Bien sûr, Scilab reconnaît bien d'autres objets comme les booléens, les polynômes, les structures, etc.

Une variable est un objet ou un ensemble d'objets réunis dans un vecteur, une matrice, une hypermatrice ou une structure. Les variables définies sont stockées dans une « pile » (zone mémoire spécialisée) dont la taille est modifiable par la fonction `stacksize`. Pour examiner les variables définies dans l'environnement, on peut utiliser la fonction `who` :

```
->who
```

```
scicos_pal          %scicos_menu
%scicos_short       %scicos_help
%scicos_display_mode          modelica_libs
scicos_pal_libs     %helps    home      SCIHOME
PWD       TMPDIR    MSDOS     SCI       guilib
sparselib           xdesslib  percentlib
polylib   intlib    elemlib   utillib   statslib
alglib    siglib    optlib    autolib   roblib
soundlib  metalib   armalib   tkscilib  tdcslib
s2flib    mtlblib   %F        %T        %z
%s        %nan      %inf      COMPILER  %gtk
```

```
 %gui      %pvm      %tk       $         %t
 %f        %eps      %io       %i        %e
 using      14696 elements  out of    5000000.
          and         54 variables out of       9231

your global variables are...

 LANGUAGE  %helps    demolist  %browsehelp
 LCC       %toolboxes          %toolboxes_dir
 using       1203 elements  out of      11000.
          and          7 variables out of        767
```

La commande `who` fournit la liste des variables utilisées ainsi que la place mémoire utilisée dans la pile (l'unité est le mot de 8 octets). On constate que même dans un Scilab fraîchement lancé, il existe des variables prédéfinies comme `%pi` (π), `%e` ($\exp(1)$), `%t` et `%f` (booléen vrai et faux), `%i` ($\sqrt{-1}$), etc. Les variables dont le nom se termine par `lib` sont les bibliothèques de fonctions prédéfinies qui seront présentées en section 6.3).

La commande `whos` est plus précise : elle fournit en outre le type des variables utilisées avec leur taille et la place mémoire occupée.

```
->whos()
Name                     Type       Size     Bytes

whos                     function             8512
x                        constant   1 by 1    24
scicos_pal               string     12 by 2   2048
%scicos_menu             list                 2440
%scicos_short            string     12 by 2   408
%scicos_help             sch        ?         62440
%scicos_display_mode     constant   1 by 1    24
modelica_libs            string     1 by 2    312
scicos_pal_libs          string     1 by 11   376
.........................
TMPDIR                   string     1 by 1    80
MSDOS                    boolean    1 by 1    16
SCI                      string     1 by 1    96
pvmlib                   library              240
timelib                  library              488
.........................
```

Cette fonction permet aussi de lister les variables définies dont on donne les premiers caractères du nom ou le type :

```
->x = 2.5;

->whos -name x  //liste les variables dont le nom commence par x
Name                    Type           Size           Bytes

x                       constant       1 by 1         24
xdesslib                library                       3008

->whos -type constant
Name                    Type           Size           Bytes

x                       constant       1 by 1         24
%scicos_display_mode    constant       1 by 1         24
%nan                    constant       1 by 1         24
%inf                    constant       1 by 1         24
%eps                    constant       1 by 1         24
%io                     constant       1 by 2         32
%i                      constant       1 by 1         32
```

Il peut être intéressant de savoir si un nom de variable est déjà utilisé ou quel est le type d'une variable :

```
->exists("x")
 ans  =

    1.

->type(x)
 ans  =

    1.

->typeof(x)
 ans  =

 constant
```

On peut libérer la place mémoire occupée par une variable en la détruisant par la commande `clear`.

Il faut noter que les variables Scilab n'ont pas besoin de déclaration de type ni d'allocation de mémoire avant leur utilisation ; elles sont créées au vol par affectation.

2.1.1 Nombres réels et complexes

Par défaut les valeurs numériques dans Scilab sont codées sur des mots double précision (64 bits). Cela signifie que les nombres sont représentés avec une précision relative de l'ordre de 10^{-16}. Donc, contrairement aux logiciels de calcul formel, les résultats obtenus par Scilab ne sont pas exacts ; toutefois, ils sont suffisamment précis pour la plupart des applications.

Comme la plupart des langages, Scilab fournit les opérations mathématiques habituelles : addition, soustraction, multiplication, division, ... et les fonctions élémentaires comme sinus, cosinus, exponentielle, etc.

```
->a = 1+1
 a  =

    2.

->x = 0.3; y = log(sin(1+x))
 y  =

  - 0.0371224
```

La syntaxe des instructions simples est <nom> = <expression>, où <nom> représente le nom sous lequel sera stocké le résultat de l'évaluation et <expression> est une expression faisant intervenir des opérateurs et/ou des fonctions. Noter l'utilisation du point virgule comme séparateur d'instructions. Les instructions se terminant par « ; » n'affichent pas leur résultat contrairement à celles terminées par « , » ou un retour à la ligne.

À défaut d'une assignation à gauche, le résultat est stocké dans la variable temporaire de nom `ans` qui est conservée tant qu'elle n'est pas redéfinie.

```
->exp(%i-2)
 ans  =

    0.0731220 + 0.1138807i
```

2.1.2 Chaînes de caractères

Pour définir une chaîne de caractères (ou string), on utilise des apostrophes « ' » ou des guillemets anglo-saxons « " » (appelés aussi « double quote »). Comme l'apostrophe représente aussi la transposition des matrices et afin de rendre les exemples plus lisibles, nous avons convenu dans ce livre de n'utiliser que la caractère « " » pour délimiter des chaînes de caractères.

```
->"Chaine = string"
 ans  =

 Chaine = string
```

Dans le cas où la chaîne contient déjà des apostrophes ou des guillemets, il faut les doubler :

```
 ->"Je m'appelle Scilab"
 ans  =

 Je m'appelle Scilab
```

Les opérations de base sur les chaînes de caractères sont la concaténation, notée par l'opérateur « + » :

```
 ->str = "Je m'appelle"+" Scilab"
 str  =

 Je m'appelle Scilab
```

et l'extraction de sous-chaîne :

```
->part(str, 14:20)
 ans  =

 Scilab
```

Dans cet exemple la fonction `part` permet d'extraire de la chaîne `str` la sous-chaîne se trouvant entre les positions 14 et 20. Comme nous le verrons plus tard, le symbole « : » permet de définir un vecteur d'indices.

Les chaînes de caractères peuvent aussi être utilisées pour définir des expressions ou des instructions du langage. La fonction `evstr` évalue l'expression définie par la chaîne de caractères qui lui est passée comme argument, tandis que la fonction `execstr` exécute une instruction donnée par une chaîne caractères :

```
->expression = "sin(3)+1"
 expression  =

 sin(3)+1

->evstr(expression)
 ans  =

    1.14112
->instruction = "z = sin(3)+1";
->execstr(instruction); z
 z  =

    1.14112
```

2.1.3 Booléens

L'objet booléen peut prendre deux valeurs : « vrai » `T` et « faux » `F`. À l'initialisation, Scilab définit deux variables booléennes `%t` et `%f`. Les résultats des opérateurs de comparaisons (« == », « > », « >= », « < », « <= », et « ~= »,), appliqués à des objets, sont des booléens :

```
->i = 0; i<10,
 ans  =

  T
->1>=2
 ans  =

  F
```

La négation se note par un « ~ » placé avant la variable booléenne.

```
->~(1>=2)
 ans  =

  T
```

Et l'on peut bien évidemment effectuer les opérations booléennes classiques.

```
->%t & %t
 ans  =

  T

->%t | %t
 ans  =

  T
```

Les variables booléennes servent essentiellement à construire des instructions conditionnelles :

```
->vrai=%t;

->if vrai then disp("Hello"),end

 Hello
```

2.1.4 Autres objets de base

D'autres objets mathématiques sont prédéfinis, comme les polynômes et les fractions rationnelles. Les objets les plus importants pour définir des objets complexes sont les structures que nous verrons en détail au paragraphe 6.1. Il existe aussi la possibilité de coder des nombres sur des entiers de 1, 2 ou 4 octets (`int8`, `int16`, `int32`). Cela est utile par exemple pour traiter des images.

Finalement, il est aussi possible à l'utilisateur, grâce au mécanisme de surcharge que nous verrons en section 6.2, de définir ses propres objets ainsi que les opérations et les fonctions correspondantes.

2.2 Objets vectoriels et matriciels

La caractéristique essentielle de Scilab est de pouvoir stocker et traiter les objets de base sous forme de tableaux 1D, 2D ou même multidimensionnels (vecteurs, matrices et hypermatrices). Cet aspect est fondamental car cela permet d'écrire du code compact, efficace et lisible.

2.2.1 Construction de vecteurs et de matrices

Une matrice est un tableau contenant certains objets de base comme les nombres entiers, réels ou complexes, les chaînes de caractères, les booléens et les polynômes. Un vecteur est un cas particulier de matrice n'ayant qu'une seule ligne ou colonne.

Les matrices et les vecteurs peuvent être définis à l'aide des opérateurs élémentaires : « [ ] », « , » et « ; ». Considérons le tableau suivant qui pourrait représenter une liste d'abonnés ; on y placerait le nom et l'âge de chaque abonné :

```
->T_abb = ["Francois","32";"Pierre","31";"Ali","76"]
 T_abb  =

!Francois  32  !
!              !
!Pierre    31  !
!              !
!Ali       76  !
```

La virgule est utilisée pour séparer les colonnes et le point-virgule pour séparer les lignes. Les crochets servent à délimiter le contenu de la matrice. Les objets que l'on utilise pour fabriquer une matrice doivent être de même type ou de types compatibles.

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

!   1.    3. !
!   4.    5. !

->P = [1,  %s+1] //concaténation d'un réél et d'un polynôme
 P  =

!   1     1 + s  !
```

On peut utiliser des matrices dans la construction d'autres matrices :

```
->B = [A, A; [0, 1, 0, 1]]
 B  =
```

```
!   1.    3.    1.    3. !
!   4.    5.    4.    5. !
!   0.    1.    0.    1. !
```

Il est aussi possible d'utiliser une notation plus proche de la représentation matricielle pour définir une matrice. Prenons le cas de `A`, on aurait pu la définir comme suit :

```
->A = [ 1.2   3
->      4    -5.06 ]
 A  =

    1.2    3.
    4.   - 5.06
```

Noter l'utilisation des blancs à la place des virgules et les retours chariots (retours à la ligne) à la place des points-virgules. Cette façon de définir les matrices, comme nous le verrons plus tard, est particulièrement utile pour la lecture des fichiers de données (voir paragraphe 4.5.4).

Il existe des fonctions qui permettent la construction des vecteurs et des matrices particulières. Par exemple, `ones`, `zeros`, et `rand` produisent respectivement des matrices contenant des uns, des zéros et des nombres aléatoires, les dimensions des matrices étant données par les arguments (`zeros(2,3)`, `rand(2,2)`, etc.) :

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

->A + 2*ones(A)
 ans  =

!   3.    5. !
!   6.    7. !
```

Pour définir un vecteur d'indices on utilise l'opérateur « : » :

```
->I = 1:5
 I  =

!   1.    2.    3.    4.    5. !
->I2 = 1:2:6
 I2  =

!   1.    3.    5. !
```

La construction de `I2` utilise la spécification d'un incrément, le pas. La syntaxe «`a:h:b`» donne un vecteur ligne contenant des valeurs allant de `a` à `b` par pas de `h`. Attention, dans le cas où le pas `h` n'a pas une valeur entière : à

cause de problèmes de précision sur certaines machines, il n'est pas sûr que l'on obtienne le bon nombre d'éléments dans le vecteur obtenu. Il est donc conseillé, pour plus de sécurité, d'utiliser des pas à valeur entières dans la construction d'un vecteur, et donc de remplacer par exemple `0:0.2:10` par `(0:2:100)/10`.

D'autres fonctions de construction de vecteurs et de matrices souvent utilisées sont `linspace`, `logspace` qui construisent des discrétisations régulières ou logarithmiques d'un intervalle et `matrix` qui permet de changer la forme d'un tableau.

Pour avoir la taille d'une matrice `M`, on utilise la fonction `size`. L'instruction `s=size(M)` donne les nombres de lignes et de colonnes de `M` dans le vecteur `s`. On obtient le même résultat en faisant `[m,n]=size(M)`.Noter que `s`, `m`, `n` sont ici des variables de type flottant et non entier. De plus, `size` peut accepter un deuxième argument :

- `size(M,1)` donne le nombre de lignes de `M` ;
- `size(M,2)` donne le nombre de colonnes de `M` ;
- `size(M,k)` donne le nombre d'éléments de la `k` ème coordonnée du tableau multidimensionnel `M` ;
- `size(M,"*")` donne le nombre total d'éléments de `M`.

2.2.2 Extraction et insertion

La notation `A(i,j)`, si les indices `i` et `j` sont des nombres, désigne l'élément (i,j) de la matrice `A`. Cette notation est utilisée aussi bien pour l'insertion que pour l'extraction. Considérons la matrice aléatoire de taille 3×3 :

```
->A=rand(3,3)
 A  =

!    .2113249     .3303271     .8497452 !
!    .7560439     .6653811     .6857310 !
!    .0002211     .6283918     .8782165 !
```

On peut extraire l'élément (2,3) de cette matrice par la commande :

```
->A(2,3)
 ans  =

     .685731
```

Pour remplacer cet élément par `0`, on fait `A(2,3)=0`.

Les indices `i` et `j` peuvent aussi être des vecteurs. Dans ce cas on désigne une sous-matrice de la matrice concernée, formée des éléments se trouvant à « l'intersection » des lignes et des colonnes sélectionnées par les vecteurs d'indice comme représenté sur la figure 2.1.

```
->A([1,2],[1,3])
 ans  =
```

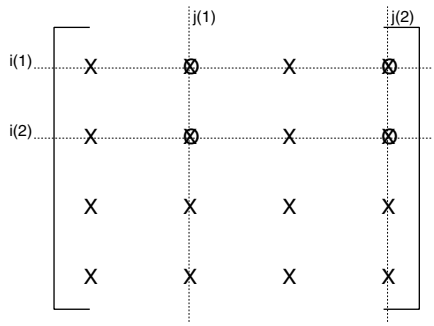


Figure 2.1 – Référencement d'une sous-matrice.

```
!    .2113249     .8497452 !
!    .7560439     .6857310 !
```

Les éléments d'un vecteur d'indices ne sont pas forcément croissants ou distincts :

```
->A([2,1],[1,1])
 ans  =

!   0.7560439    0.7560439 !
!   0.2113249    0.2113249 !
```

Le symbole « : » peut être utilisé à la place d'un indice pour désigner « tout » (toutes les lignes ou toutes les colonnes, en fonction du placement).

```
->A(1,:)
 ans  =

!    .2113249     .3303271     .8497452 !
```

On peut ainsi insérer une sous-matrice de la façon suivante :

```
->A([1,2],[1,3])=ones(2,2)
 A  =

!   1.           .3303271    1.        !
!   1.           .6653811    1.        !
!    .0002211    .6283918     .8782165 !
```

Ici, on a inséré une matrice dont tous les éléments ont la même valeur `1`. On peut aussi effectuer cette opération avec la commande `A([1,2],[1,3])=1`. Cela peut paraître contradictoire car `A([1,2],[1,3])` et `1` n'ont pas la même

taille, mais c'est une facilité de programmation. Pour la même raison, on accepte l'addition, la soustraction et la comparaison d'une matrice avec un objet scalaire, par exemple :

```
->B=A+2
 B  =

!   3.           2.3303271    3.         !
!   3.           2.6653811    3.         !
!   2.0002211    2.6283918    2.8782165 !

->A==1
 ans  =

! T F T !
! T F T !
! F F F !
```

Bien entendu la même chose marche aussi pour d'autres types de matrices, par exemple pour le tableau des abonnés `T_abb` (défini page 17) qui est une matrice de chaînes de caractères. Supposons que l'on veuille rajouter une colonne à `T_abb` pour indiquer la ville de naissance de chaque abonné et que tous les abonnés actuels sont nés à Paris :

```
->T_abb(:,3)="Paris"
 T_abb  =

!Francois  32   Paris  !
!                      !
!Pierre    31   Paris  !
!                      !
!Ali       76   Paris  !
```

L'insertion d'une matrice vide « `[]` » peut être utilisée pour supprimer une ou plusieurs lignes (ou colonnes) d'une matrice. Par exemple pour supprimer les deux premières lignes de `B`, on peut faire

```
->B([1,2],:)=[]
 B  =

!   2.0002211    2.6283918    2.8782165 !
```

Cette instruction est équivalente à `B=B(3:$,:)` car « `$` » désigne le dernier élément. Le symbole « `$` » permet de définir des opérations sur des matrices sans avoir à calculer leur taille (ce qui pourrait être fait par la fonction `size`). Par exemple, pour échanger la première et la dernière ligne de la matrice `A`, on peut écrire :

```
A([1,$],:)=A([$,1],:)
```

Le symbole « $ » est aussi utilisé pour insérer des éléments dans une matrice. Par exemple, si l'on revient à notre tableau d'abonnés, voici comment ajouter un abonné à T_abb :

```
->T_abb($+1,:)=["Tina","6","Paris"]
 T_abb  =

!Francois  32   Paris  !
!                      !
!Pierre    31   Paris  !
!                      !
!Ali       76   Paris  !
!                      !
!Tina      6    Paris  !
```

Les insertions et les extractions permettent d'effectuer des opérations complexes avec un nombre réduit d'instructions. Par exemple, pour trouver l'âge de Pierre, une seule instruction suffit :

```
->T_abb(find(T_abb(:,1)=="Pierre"),2)
 ans  =

 31
```

Pour comprendre le fonctionnement de cette instruction, on procède pas à pas. En premier, on a

```
->T_abb(:,1)
 ans  =

!Francois  !
!          !
!Pierre    !
!          !
!Ali       !
!          !
!Tina      !
```

qui est un vecteur de chaînes de caractères. La comparaison de ce vecteur avec "Pierre" donne le vecteur booléen :

```
->T_abb(:,1)=="Pierre"
 ans  =

! F !
! T !
! F !
! F !
```

La fonction `find` retourne un vecteur de nombres qui contient les indices des `T` (vrai) dans un vecteur de booléens. Dans ce cas il n'existe qu'une seule valeur vraie qui se trouve en deuxième place. Le résultat de `find` est donc 2. Donc, `T_abb(find(T_abb(:,1)=="Pierre"),2)` est `T_abb(2,2)` ce qui est le résultat désiré.

L'insertion et l'extraction pour les vecteurs peuvent bien évidemment être définies par un seul indice `V(i)`

```
->V=[1 4 12 8];
->V([1 2])
 ans  =

!   1.    4. !
->V([4 6])=[-1 -2]
 V  =

!   1.    4.    12.  - 1.    0.  - 2. !
```

On note ici, que le vecteur `V` est automatiquement redimensionné lorsque l'on insère des éléments au delà de sa taille courante.

La notation avec un seul indice est aussi valide pour les objets matriciels. Le plus souvent l'usage d'un seul indice pour les matrices revient à mettre en correspondance la matrice avec un vecteur colonne formé par la concaténation des colonnes de la matrice.

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

->A(3)
 ans  =

    2.
->A([1,3])
 ans  =

!   1. !
!   2. !
->A(3)=10
 A  =

!   1.    10. !
!   3.    4.  !
-> A(6)=33
       !-error    21
invalid index
```

Il faut noter que la syntaxe `X=A(:)` retourne un vecteur colonne `X` formé par la concaténation des colonnes de la matrice `A`.

Pour finir avec l'insertion et l'extraction, notons que l'on utilise le plus souvent des variables de type "nombre flottant" comme indice. Cependant, il est aussi possible d'utiliser des types entiers. Dans le cas où un indice n'a pas une valeur entière, c'est la partie entière qui est prise en compte pour l'indexation.

2.2.3 Opérations vectorielles et matricielles

La plupart des opérateurs fonctionnant sur les objets de base fonctionnent aussi sur les vecteurs et les matrices. Par exemple l'addition et la soustraction.

```
->a=[1:3;2:4]
 a  =

!   1.    2.    3. !
!   2.    3.    4. !

->a+a
 ans  =

!   2.    4.    6. !
!   4.    6.    8. !
```

Mais aussi pour les fonctions de base

```
->sin(a)
 ans  =

!   0.8414710    0.9092974    0.1411200 !
!   0.9092974    0.1411200  - 0.7568025 !
```

Ces extensions restent valables pour d'autres types de données matricielles

```
->a=["s","fr";"wer","sdf"]
 a  =

!s    fr   !
!          !
!wer  sdf  !

->a+a
 ans  =

!ss      frfr    !
!                !
!werwer  sdfsdf  !

->part(a,1)
 ans  =

!s  f  !
```

```
!      !
!w  s  !

->length(a)
 ans  =

!   1.    2. !
!   3.    3. !
```

Ici, `length` donne la longueur d'une chaîne de caractères.

Mais quand il s'agit des opérations de type multiplication, la situation est un peu différente car la multiplication de deux matrices, avec les notations habituelles, ne signifie pas la multiplication élément par élément mais la multiplication au sens mathématique du produit matriciel. C'est pour cela que Scilab utilise deux opérateurs distincts pour représenter la multiplication matricielle : « `*` » et « `.*` ». Le point placé avant l'opérateur indique que l'opération est effectuée élément par élément. Les autres opérations de ce type sont « `/` » et « `./` » pour la division à droite, « `\` » et « `.\` » pour la division à gauche, « `^` » et « `.^` » pour l'élévation à la puissance :

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

!   1.     4.     9.  !
!   16.    25.    36. !

->A.^(1/2) //la racine carrée des éléments de A
 ans  =

!   1.    1.4142136    1.7320508 !
!   2.    2.236068     2.4494897 !
->2 .^ (0:8)
 ans  =

!   1.   2.   4.   8.   16.   32.   64.   128.   256. !
```

2.2.4 Matrices creuses

Dans certaines applications, on rencontre des grosses matrices qui contiennent un faible pourcentage d'éléments non nuls. Pour ce type de matrices, Scilab utilise un codage spécial dit « creux » où seuls les éléments non nuls sont stockés en mémoire. La fonction `sparse` permet de définir une matrice creuse à partir de la liste de ses éléments non nuls. La manipulation des matrices creuses se fait de manière identique à celle des matrices pleines. La plupart du temps une opération faisant intervenir simultanément des matrices creuses et des matrices pleines fournit comme résultat une matrice pleine.

```
->sp=sparse([1,2;4,5;3,10],[1,2,3],[4,10])
 sp  =

(    4,   10) sparse matrix

(    1,    2)        1.
(    3,   10)        3.
(    4,    5)        2.

->1+sp
 ans  =

!  1.   2.   1.   1.   1.   1.   1.   1.   1.   1. !
!  1.   1.   1.   1.   1.   1.   1.   1.   1.   1. !
!  1.   1.   1.   1.   1.   1.   1.   1.   1.   4. !
!  1.   1.   1.   1.   3.   1.   1.   1.   1.   1. !
```

Seules les matrices de nombres et de booléens peuvent être codées sous forme creuse.

2.2.5 Une structure simple

On désigne ici par « structure » une structure de données composite formée en rassemblant dans une même variable plusieurs objets pouvant être de type différent.
Nous verrons plus en détail (section 6.1) les différents types de structures existant dans Scilab. La structure la plus simple s'appelle `list`.

```
->struct=list("une chaine",rand(2,2))
 struct  =


       struct(1)

 une chaine

       struct(2)

!   0.5715097    0.1205854 !
!   0.0549629    0.0143620 !

->struct2=list(struct,int32(12))
 struct2  =


       struct2(1)


        struct2(1)(1)
```

```
 une chaine

        struct2(1)(2)

!   0.5715097    0.1205854 !
!   0.0549629    0.0143620 !

       struct2(2)

 12
```

L'insertion et l'extraction d'un élément dans les `list` sont similaires à celles des vecteurs.

```
->struct2(2)=eye(2,2)
 struct2  =


       struct2(1)


        struct2(1)(1)

 une chaine

        struct2(1)(2)

!   0.5715097    0.1205854 !
!   0.0549629    0.0143620 !

       struct2(2)

!   1.    0. !
!   0.    1. !

->struct($+1)=12
 struct  =


       struct(1)

 une chaine

       struct(2)

!   0.5715097    0.1205854 !
!   0.0549629    0.0143620 !

       struct(3)
```

```
    12.
```

Toutefois, contrairement au cas des vecteurs, on ne peut pas insérer plusieurs éléments dans une `list` avec une seule instruction. En revanche, l'extraction peut se faire comme suit :

```
->[p,q]=struct(1:2)
 q  =

!   0.5715097    0.1205854 !
!   0.0549629    0.0143620 !
 p  =

 une chaine
```

Pour supprimer un élément d'une liste on y insère `null()` :

```
->struct(2)=null()
 struct  =


       struct(1)

 une chaine

       struct(2)

    12.
```

N'importe quel élément peut être supprimé. Il n'est possible de rajouter un élément qu'au début ou à la fin d'une `list`. On a déjà vu un exemple d'ajout en fin de `list` en utilisant l'indice `$+1`. Pour rajouter un élément au début, on procède comme suit :

```
->struct(0)="le premier"
 struct  =


       struct(1)

 le premier

       struct(2)

 une chaine

       struct(3)

    12.
```

On notera que la `list` vide se définit par `list()`.

2.2.6 La vectorisation

Dans Scilab, la plupart du temps on manipule des vecteurs et des matrices. Les opérateurs et les fonctions élémentaires sont conçus pour favoriser ce type de manipulation et, de manière plus générale, pour permettre la vectorisation des programmes. Certes, le langage Scilab contient des instructions conditionnelles (`if`, `select`), des boucles (`while`, `for`) et la programmation récursive, mais la vectorisation permet de limiter le recours à ces fonctionnalités qui ne sont jamais très efficaces dans le cas d'un langage interprété. Les surcoûts d'interprétation peuvent être très pénalisants par rapport à ce que ferait un programme C ou FORTRAN compilé lorsque l'on effectue des calculs numériques. Il faut donc veiller à réduire autant que possible le travail d'interprétation en vectorisant les programmes.

Par exemple, pour représenter graphiquement la fonction $\cos(2x^2)$ entre 0 et 5 avec une précision de 1000 points, on peut définir sans boucle le vecteur contenant les points à afficher, de la manière suivante :

```
->x=[0:999]*5/1000;
```

On peut, par la suite, utiliser ce vecteur pour évaluer la fonction et tracer la courbe (figure 2.2) :

```
->y=cos(2*x.^2);

->plot2d(x,y)
```

Dans cet exemple le gain d'efficacité par rapport à une programmation scalaire est un facteur de l'ordre de 80 !

On peut trouver, sans aucune boucle, les points où la fonction $\cos(2x^2)$ est extrémale en utilisant le fait que la dérivée aux points extrêmes change de signe. On calcule d'abord le vecteur `dy` dont les composantes sont $y_{i+1} - y_i$, puis le vecteur `p` contenant $(y_{i+1} - y_i)(y_{i+2} - y_{i+1})$ et enfin on cherche les composantes négatives de `p` :

```
->dy=y(2:$)-y(1:$-1);

->p=dy(1:$-1).*dy(2:$);

->x_ext=x(find(p<=0))
 x_ext  =
         column 1 to 8
! 1.25  1.765  2.165  2.5  2.795  3.065  3.31  3.54 !

         column  9 to 15
! 3.755  3.96  4.15  4.335  4.515  4.685  4.85 !
```

Le symbole « `$` » désigne ici 1000, c'est-à-dire l'indice du dernier élément de `y` et 999 dans le cas de `dy`. Le résultat de `p<=0` est un vecteur de booléens dont

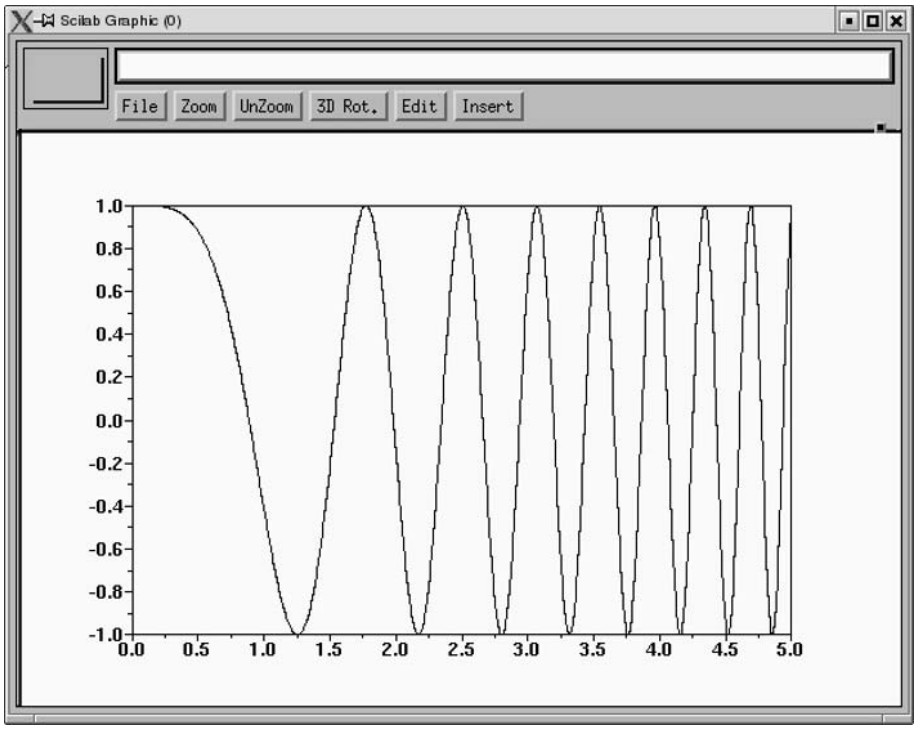


Figure 2.2 – Exemple d'un `plot2d`.

les indices correspondant aux éléments ayant la valeur `T` sont déterminés par la fonction `find`.

Les graphiques 3D sont souvent réalisés en utilisant des opérations matricielles. Par exemple, la surface $f(x, y) = 10 \sin(x) \cos(y)$, $0 \leq x < \pi$, $0 \leq y < 2\pi$, peut être dessinée comme suit (figure 2.3) :

```
->x=linspace(1,%pi,25);

->y=linspace(1,2*%pi,50);

->plot3d(x,y,10*sin(x)'*cos(y))
```

L'opérateur « ' » correspond à la transposition. De façon générale, cette opération remplace l'élément i, j d'une matrice par l'élément j, i. Cette opération appliquée au vecteur ligne `sin(x)` le convertit en un vecteur colonne. `sin(x)'*cos(y)` est le produit d'un vecteur colonne de taille 100 par un vecteur ligne de taille 100, ce qui est la matrice à 100 lignes et 100 colonnes définie par $[\sin(x_i) \cos(y_j)]_{i,j}$.

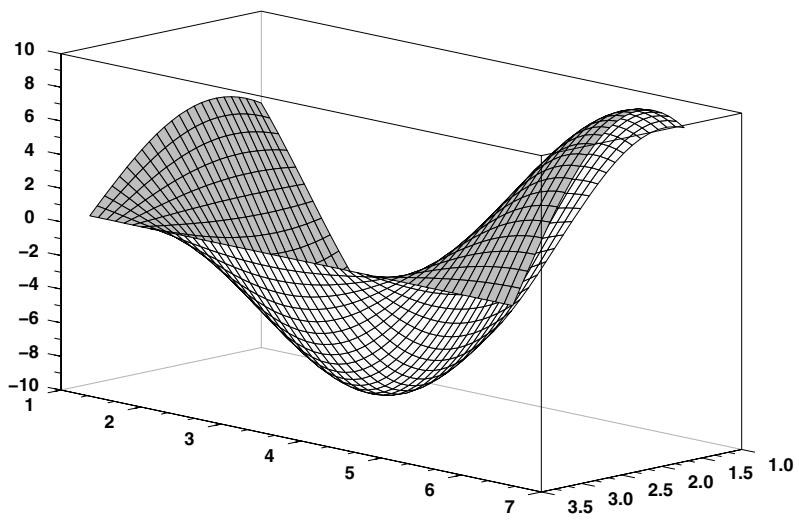


Figure 2.3 – `plot3d` de la fonction $10\sin(x)\cos(y)$.

Chapitre 3

Programmation de base

3.1 Script Scilab

Un programme Scilab est une suite d'instructions. Ces instructions peuvent être placées dans un fichier (que l'on appelle un *script*) à l'aide d'un éditeur de texte : l'éditeur associé `scipad` ou un autre.[1]

Si par exemple le fichier s'appelle `toto.sce` et contient :

```
a=[1 2 3
   4 5 6
   7 8 9];
b=2,
```

on peut alors exécuter le script avec l'instruction `exec("toto.sce")` . Pour la fonction `exec`, ainsi que pour un certain nombre d'autres dans Scilab, il est possible d'utiliser la syntaxe plus simple `exec toto.sce`. On obtient :

```
->exec("toto.sce")

->a=[1 2 3
->   4 5 6
->   7 8 9];

->b=2,
 b  =

    2.
```

et tout se passe alors comme si les lignes écrites dans le fichier `toto.sce` avaient été introduites une par une au clavier.

[1] Il faut noter que pour les utilisateurs préférant Emacs, il existe un mode Scilab pour cet éditeur.

Un choix d'options permet de contrôler le niveau d'affichage lors de l'exécution. Il est possible de contrôler l'affichage des instructions ainsi que leurs résultats :

```
->exec("toto.sce",2)

 b  =

    2.
```

En particulier, positionner l'option à `-1` supprime tout affichage (faire `help mode`). La valeur par défaut est 0 si l'instruction `exec` est terminée par un point-virgule et 2 sinon.

Lors du lancement de Scilab, un fichier script particulier (`scilab.star`) est exécuté pour initialiser un certain nombre de variables et pour charger des bibliothèques de fonctions. De plus Scilab exécute le contenu des fichiers `.scilab` ou `scilab.ini` (s'ils existent) se trouvant dans le répertoire utilisateur donné par la variable Scilab `SCIHOME` puis dans le répertoire où Scilab est lancé. Sous Windows, la localisation du répertoire `SCIHOME` dépend de la version de l'OS ; sous Unix ce répertoire se trouve dans `~/.Scilab/scilab-xxxx` où `xxxx` caractérise la version de Scilab.

3.2 Boucles et instructions de contrôle

Il existe plusieurs manières de construire des boucles et des instructions de contrôle. On en a déjà vu quelques-unes.

3.2.1 Instruction `for`

L'instruction `for` est la façon la plus courante d'effectuer des boucles. Cette instruction exécute les instructions qu'elle contient (jusqu'au mot clé `end`) un nombre déterminé de fois inconditionnellement. Par exemple[2] :

```
->Ages = [33 56 8 75 24];
->somme = 0; for age=Ages, somme = somme+age; end

->moyenne = somme/length(Ages)
 moyenne  =

    39.2
```

Comme `Ages` est un vecteur ligne, la variable `age` prend ici comme valeurs successives les éléments de `Ages` et le contenu de l'instruction `for`, qui est formé de toutes les instructions comprises entre `for` et `end`. La boucle est exécutée autant de fois qu'il y a d'éléments dans `Ages`. Si à la place de `Ages` on avait une

[2] La manière efficace de calculer la somme du vecteur `ages` est, bien entendu, `sum(ages)`.

matrice, `age` serait un vecteur colonne qui prendrait comme valeurs successives les colonnes de cette matrice. Si à la place on avait une `list`, on aurait alors les éléments de la `list`. Donnons un exemple :

```
->mystruct =l ist(list(),"coucou",3);

->for elmt=mystruct
->  disp(typeof(elmt))
->end

 list

 string

 constant
```

On obtient l'affichage du type de chacun des éléments de la `list mystruct`.

3.2.2 Boucle `while`

La boucle `while` exécute son contenu tant que son argument est un booléen vrai. L'exemple ci-dessous calcule par dichotomie le plus petit nombre Scilab qui ajouté à `1` donne un résultat numériquement différent[3] de `1`.

```
->x=1;
->while 1+x>1, x=x/2;end
->x
 x  =

    1.110D-16
->(1+x)==1
 ans  =

  T
```

Les boucles `for` et `while` peuvent être terminées prématurément par l'instruction `break`.

```
->for i=1:1000
->  if i==100 then break;end
->end

->i
 i  =

    100.
```

[3] Scilab fait ses calculs numériques en utilisant l'arithmétique double précision. La précision relative de la machine est définie comme étant le plus petit nombre ϵ tel que 1 et $1+\epsilon$ ont des représentations distinctes en Scilab.

```
->for i=1:1000,end

->i
 i  =

    1000.
```

On remarque ici qu'à la fin, normale ou terminée par un `break`, de l'exécution d'une boucle la variable de boucle `i` subsiste ; il est ainsi possible de connaître quelle valeur de `i` a provoqué la fin de la boucle ou de reprendre sa valeur pour l'utiliser dans la suite des instructions.

L'instruction `continue` permet de passer directement à l'itération suivante d'une instruction `for` ou `while` en sautant les instructions jusqu'au `end`.

```
->for i=1:size(a,'*')
->  if a(i)==0 then continue,end
->  a(i)=1/a(i)
->end
```

3.2.3 Instruction de contrôle `if`

L'instruction de contrôle `if` permet, comme dans l'exemple précédent, d'exécuter conditionnellement des instructions

```
T_abb = [1 2 3 4 5 6 7 8 9]';
i = 1;str = " "
while str ~= "q"
  disp(T_abb(i,:));
  str = input("Sortir(q), Suivant(s), Precedent(p)?","string")
  if str == "s" then
     i = min(i+1, size(T_abb,1));
  elseif str == "p" then
     i = max(i-1, 1);
  else
    disp("saisie incorrecte ")
  end
end
```

Fichier source : `scilab/if.sce`

Noter l'utilisation de `if elseif` qui permet de spécifier plusieurs conditions. Seules les instructions correspondant à la première condition vérifiée sont exécutées.

3.2.4 Instruction de contrôle `select-case`

On aurait aussi pu programmer l'exemple précédent en utilisant la construction :

```
i = 1;
while %t
  disp(T_abb(i,:))
  select input("Sortir(q), Suivant(s), Precedent(p)?", "string")
  case "s" then
     i = min(i+1, size(T_abb,1));
  case "p" then
     i = max(i-1, 1);
  case "q" then
     break;
  else
     disp("saisie incorrecte ")
  end
end
```

Fichier source : `scilab/select.sce`

La fonction `input` (voir la section 4.2.1) permet ici de demander à l'utilisateur de fournir une réponse.

3.2.5 Instruction de contrôle `try-catch`

Dans certains cas, il est souhaitable d'organiser le déroulement d'un programme en fonction du déclenchement possible d'une erreur. Pour cela on peut utiliser l'instruction de contrôle `try-catch` :

```
try
  fid=mopen("foo","r");//ouvre le fichier foo
  titre=mgetl(fid,1);// lecture de la première ligne
catch
  mprintf("Le fichier ""foo"" n'existe pas,\n"+..
          "titre prend la valeur par défaut");
  titre="essai";
end
```

Fichier source : `scilab/try.sce`

Dans l'exemple ci-dessus, si le fichier `foo` n'existe pas, les instructions suivantes du bloc `try` sont omises et l'évaluation reprend avec les instructions contenues dans le bloc `catch`. D'autres méthodes de contrôle d'erreur sont décrites en section 6.4.

3.3 Fonctions Scilab

Le langage Scilab permet de définir de nouvelles fonctions. Ce qui distingue une fonction d'un script est que celle-ci a un environnement local qui communique avec l'extérieur à travers des arguments d'entrée et de sortie.

La définition d'une fonction commence par le mot clé `function` immédiatement suivi par la description de la syntaxe d'appel de la fonction, puis des instructions à effectuer et se termine par le mot clé `endfunction`.

3.3.1 Définir une fonction en ligne

Les fonctions peuvent être définies directement en ligne :

```
->function y=sq(x) //définition de la syntaxe d'appel
->y=x^2 // instruction
->endfunction // fin de la définition

->sq(3) //appel de la fonction
  ans  =

    9
```

On aurait pu définir cette fonction en utilisant la fonction `deff` :

```
->deff("y=sq(x)","y=x^2")
->sq(3) //appel de la fonction
  ans  =

    9
```

Le premier argument de `deff` est une chaîne de caractères donnant la syntaxe d'appel, le second est un vecteur de chaînes donnant la suite des instructions à exécuter. Ce mode de définition est indispensable quand on doit écrire une fonction dont la définition dépend de variables calculées en cours de déroulement d'un programme.

Lors de l'exécution des fonctions, contrairement au cas des scripts, la visualisation automatique des résultats est inhibée, et ce que les instructions se terminent ou non par un point-virgule « `;` ». Pour afficher un résultat à l'écran dans une fonction, il faut utiliser les fonctions d'entrée-sortie décrites en section 4.2.1. Il est aussi possible de modifier explicitement le mode d'affichage avec la fonction `mode`.

3.3.2 Définition dans un fichier

Le plus souvent, les fonctions Scilab sont définies dans des fichiers. Cela permet à l'utilisateur d'avoir à sa disposition toute la puissance de son éditeur de texte favori pour éditer sa fonction.

Par exemple la fonction `mafonc` ci-dessous que l'on a écrit dans un fichier appelé `mafonc.sci`.

L'extension `.sci` est en général choisie lorsqu'il s'agit d'un fichier contenant une ou plusieurs fonctions. L'extension `.sce` d'un autre côté est à préférer pour les fichiers scripts. Mais en fait il n'y pas de réelle différence entre les deux types de fichier, dans les deux cas la fonction `exec` évalue les instructions contenues dans le fichier. Il est même possible de définir un fichier contenant simultanément des définitions de fonctions et des instructions. Le choix de l'extension n'a d'importance que pour les opérations "glisser déposer" et pour l'exécution automatique sous Windows.

```
function [v,ind]=mafonc(v,op)
// Cette fonction reçoit :
// - un vecteur v
// - une chaîne de caractères op indiquant
//   l'opération à effectuer sur v.
// En sortie,
// -  v est trié
// -  ind est le vecteur d'indices correspondant.
  if v==[] then return;end
  select op
  case "aleatoire" then
    // on mélange de manière aléatoire les éléments de v
    [junk,ind]=gsort(rand(v))
    v=v(ind)
  case "decroissant" then
    // on trie les éléments de v dans l'ordre décroissant
    [v,ind]=gsort(v)
  case "croissant" then
    [v,ind]=gsort(v)
    v=v($:-1:1)
    ind=ind($:-1:1)
  else
    error("opération non prévue !");
  end
endfunction
```

Fichier source : `scilab/mafonc.sci`

On voit sur cet exemple (et aussi sur la fonction `gsort`) que les fonctions peuvent avoir plusieurs arguments de sortie. La fonction `gsort` effectue le tri d'un vecteur et renvoie les valeurs triées ainsi qu'un vecteur de permutation.

Noter l'utilisation de « `//` » pour indiquer les commentaires ainsi que l'usage de la fonction `error` pour éditer un message d'erreur et déclencher le mécanisme de gestion d'erreur qui sera décrit en section 6.4.

On peut alors charger la fonction `mafonc` comme suit[4] :

```
->exec  mafonc.sci
```

On peut maintenant par exemple utiliser la fonction `mafonc` pour réordonner la liste des abonnés `T_abb` définie page 17 :

```
->T_abb  =

!Francois  32   Paris   !
!                       !
!Pierre    31   Paris   !
!                       !
!Ali       76   Paris   !
!                       !
!Tina      6    Paris   !
```

en mettant les noms dans l'ordre alphabétique comme suit :

```
->[y,ind]=mafonc(T_abb(:,1),"croissant");

->T_abb(ind,:)
 ans  =

!Ali       76  Paris  !
!                     !
!Francois  32  Paris  !
!                     !
!Pierre    31  Paris  !
!                     !
!Tina      6   Paris  !
```

Noter que le ré-ordonnancement en fonction des âges des abonnés ne peut pas être fait de la même manière car les âges sont codés comme des chaînes de caractères et non comme des nombres. Il faut préalablement les convertir en nombres. Pour cela, on peut utiliser la fonction `evstr` :

```
->[y,ind]=mafonc(evstr(T_abb(:,1)),"croissant");
->ages=T_abb(ind,1)
 ages  =

!   6.  !
!   31. !
!   32. !
!   76. !
```

[4]Sous Windows, le chargement peut s'effectuer simplement en faisant glisser l'icône du fichier dans la fenêtre Scilab.

Noter l'appel de la fonction avec un seul argument de sortie ; dans ce cas, la sortie correspond au premier argument de sortie de la définition de la fonction. Nous verrons plus en détail en section 3.4.2 l'utilisation d'un nombre plus petit d'arguments (aussi bien en entrée qu'en sortie) que le nombre d'arguments dans la définition de la fonction.

Il est possible de définir des sous-fonctions à l'intérieur d'une fonction. En utilisant cette facilité la fonction `mafonc` ci-dessus peut-être réécrite sous la forme :

```
function [v,ind] = mafonc(v,op)
  if v == [] then return;end
  //définition des sous-fonctions
  function [v,ind] = do_aleatoire(v)
    [junk,ind] = gsort(rand(v))
    v = v(ind)
  endfunction

  function [v,ind] = do_decroissant(v)
    [v,ind] = gsort(v),
  endfunction

  function [v,ind] = do_croissant(v)
    [v,ind] = gsort(v)
    v = v($:-1:1)
    ind = ind($:-1:1)
  endfunction

  select op
  case "aleatoire" then
    [v,ind] = do_aleatoire(v)
  case "decroissant" then
    [v,ind] = do_decroissant(v)
  case "croissant" then
    [v,ind] = do_croissant(v)
  else
    error("opération non prévue !");
  end
endfunction
```

Fichier source : `scilab/mafonc3.sci`

3.4 Exécution d'une fonction

On a vu que la commande `exec`, ayant comme argument le nom d'un fichier, exécute le contenu du fichier. Cette commande peut aussi avoir comme argument une variable de type `function`. Dans ce cas, le contenu de la fonction est

exécuté comme s'il s'agissait d'un script. Les arguments d'entrée et de sortie de la fonction sont ignorés et sont supposés préexister dans l'environnement.

En utilisant cette facilité, on peut réécrire la fonction `mafonc` comme suit :

```
function [v,ind] = mafonc1(v,op)
  if v ==[ ] then return;end
  //définition des sous-fonctions
  function do_aleatoire()
    [junk,ind] = gsort(rand(v));
    v = v(ind);
  endfunction

  function do_decroissant(),[v,ind] = gsort(v);endfunction

  function do_croissant()
    [v,ind] = gsort(v);
    v = v($:-1:1);
    ind = ind($:-1:1);
  endfunction

  select op
  case "aleatoire" then
    exec(do_aleatoire)
  case "decroissant" then
    exec(do_decroissant)
  case "croissant" then
    exec(do_croissant)
  else
    error("operation non prevue !");
  end
endfunction
```

Fichier source : `scilab/mafonc1.sci`

Ces fonctions `do_decroissant`, `do_croissant` et `do_aleatoire` auraient pu, bien sûr, être définies dans un autre fichier que celui où se trouve `mafonc1`. Ce qui est important c'est qu'au moment où l'on exécute `mafonc1`, ces fonctions soient chargées dans Scilab.

Noter que l'utilisation de `exec` à la place des appels à des fonctions permet de se dispenser d'avoir à gérer des listes d'appel, à condition bien entendu que les noms de variables correspondent.

Des points-virgules (« ; ») sont placés à la fin de chaque instruction dans la définition des fonctions `do_decroissant`, `do_croissant` et `do_aleatoire`, pour éviter l'affichage des résultats à l'écran.

Cette réécriture de `mafonc` permet de définir les opérations à effectuer pour chaque requête en dehors de `mafonc`. Mais elle ne permet pas de définir des nouvelles requêtes sans modifier le code de la fonction `mafonc`. Pour cela on peut réécrire `mafonc1` comme suit :

```
function [v,ind] = mafonc2(v,op)
  if v == [] then return;end
  i = find(list_operation == "do_"+op)
  if i  == [] then
    error("La requete "+op+" n'est pas disponible.")
  else
    i = i(1)
    execstr("exec("+list_operation(i)+",-1)")
  end
endfunction
```

Fichier source : `scilab/mafonc2.sci`

Dans ce cas, le vecteur de chaînes de caractères `list_operation` doit contenir la liste des requêtes existantes. Pour chaque élément « `toto` » de `list_operation`, il faut qu'une fonction « `do_toto` » soit chargée dans Scilab. Dans ce cas on peut complètement adapter la fonction `mafonc` en ajoutant ou en supprimant des requêtes dans `list_operation`. La fonction `execstr` exécute une instruction donnée par une chaîne de caractères.

3.4.1 Fonctions Scilab et primitives

Scilab contient des centaines de fonctions préprogrammées pour résoudre des problèmes mathématiques spécifiques. Mais toutes les fonctions ne sont pas écrites en langage Scilab. Les fonctions très élémentaires et les fonctions pour lesquelles la vitesse d'exécution est un facteur crucial sont codées en C ou en FORTRAN et interfacées avec Scilab. On appelle ces fonctions des primitives. On verra dans le chapitre 7 comment on peut ajouter de nouvelles primitives à Scilab. L'utilisateur n'a pas en général à distinguer une primitive d'une fonction. Il est cependant possible de les distinguer avec les fonctions `type` et `typeof` :

```
->type(sin)
 ans  =

    130.
->typeof(sin)
 ans  =

 fptr

->function y=foo(x),y=sin(x),endfunction
->type(foo)
 ans  =

    13.
->typeof(foo)
 ans  =

 function
```

3.4.2 Les variables dans les fonctions

En général, lorsque l'on appelle une fonction, on passe comme arguments d'entrée toutes les variables dont la fonction a besoin pour effectuer ses calculs. Les résultats des calculs sont alors affectés aux variables de sortie.

Cependant, il est aussi possible de référencer une variable dans une fonction même si celle-ci ne se trouve pas dans la liste d'appel ; il suffit qu'elle soit définie dans l'espace des variables de la fonction appelante (encore appelé contexte appelant). Considérons une fonction qui vérifie l'existence d'un nom dans la liste des abonnés `T_abb` (définie page 17).

```
function bool = abonne_existe(nom)
  bool=or(nom==T_abb(:,1))
endfunction
```

Fichier source : `scilab/abonne_existe.sci`

Cette fonction renvoie un booléen vrai si l'abonné `nom` existe ; sinon elle retourne un booléen faux (la fonction `or` renvoie le booléen vrai (`T`) si au moins l'un des éléments de son argument l'est).

Noter ici que la variable `T_abb` n'est pas passée comme argument de la fonction. Donc cette variable n'est disponible qu'en « lecture » à l'intérieur de cette fonction. Si la fonction contenait une instruction d'assignation sur la variable `T_abb` une nouvelle variable locale `T_abb` serait créée sans tenir compte de la variable existant dans le contexte appelant. Cette variable serait perdue à la sortie de la fonction. Dans tous les cas, la variable `T_abb` d'origine ne sera pas modifiée (sauf, de manière volontaire, par utilisation de `resume` que l'on verra plus tard, à la page 49).

Supposons maintenant que l'on veuille définir une fonction pour ajouter ou supprimer un abonné dans `T_abb`. Bien entendu, on peut mettre `T_abb` dans la liste des variables d'entrée et de sortie de la fonction. Cependant, on peut aussi déclarer `T_abb` comme une variable globale. Dans ce cas, `T_abb` est stockée dans un espace de variables particulier appelé espace des variables globales. Toute modification faite sur la variable `T_abb` est conservée. La fonction suivante (qui n'a pas de variable de sortie) permet par exemple de supprimer un abonné dans `T_abb`.

```
function supprime_abonne(nom)
  global T_abb
  n=find(T_abb(:,1)==nom);
  T_abb(n,:)=[]
endfunction
```

Fichier source : `scilab/supprime_abonne.sci`

L'instruction `global T_abb` permet de préciser que la variable `T_abb` réside dans l'espace des variables globales. Cette instruction est mise dans la fonction mais doit être AUSSI exécutée dans l'environnement courant pour que la variable `T_abb` y soit connue.

```
->global T_abb

->T_abb
 T_abb  =

!Francois  32  Paris  !
!                     !
!Pierre    31  Paris  !
!                     !
!Ali       76  Paris  !
!                     !
!Tina      6   Paris  !

->supprime_abonne("Ali")

->T_abb
!Francois  32  Paris  !
!                     !
!Pierre    31  Paris  !
!                     !
!Tina      6   Paris  !
```

L'exemple ci-dessus suppose que la variable `T_abb` préexiste dans l'espace des variables. L'instruction `global T_abb` a alors une double action : elle transfère la variable `T_abb` de l'espace des variables standard vers l'espace des variables globales et elle indique que cette variable devra ultérieurement être recherchée dans l'espace global.

L'appel d'une fonction peut se faire avec moins d'arguments que ceux spécifiés dans sa définition. Par exemple la fonction définie par :
`function [x,y,z]=fct(a,b)`,
peut être appelée avec la syntaxe `x=fct(a,b)` ou `[x,y]=fct(a)`. Dans le premier cas les variables `y` et `z` ne sont pas utilisées (même si elles sont effectivement calculées par `fct`). Dans le second cas, l'appel est correct tant que la variable `b` n'est pas utilisée dans `fct`. Si par contre `b` est utilisée et s'il n'y a pas de variable `b` dans l'environnement courant, on obtient un message d'erreur.

Il peut donc être utile de savoir avec combien de paramètres d'entrée et de sortie la fonction est appelée. Cela se fait avec la commande `argn`. Typiquement, la première instruction de `fct` pourrait être `[lhs,rhs]=argn()`. L'argument `lhs` donne le nombre de paramètres demandés en sortie et `rhs` donne le nombre de paramètres d'entrée effectivement passés à la fonction. Ainsi, avec l'appel `x=fct(a,b)`, on aurait `lhs` égal à 1 et `rhs` égal à 2, alors qu'avec l'appel `[x,y]=fct(a)`, on aura `lhs` égal à 2 et `rhs` égal à 1.

Considérons l'exemple suivant qui permet d'ajouter un abonné à `T_abb` :

```
function ajout_abonne(nom,age,ville)
  global T_abb
  if argn(2)==2 then ville="Paris";end
```

```
  if typeof(age)~="string" then age=string(age);end
  T_abb($+1,:)=[nom,age,ville]
endfunction
```

Fichier source : `scilab/ajout_abonne.sci`

Dans cette fonction le troisième argument `ville` est optionnel, c'est-à-dire que l'on peut appeler cette fonction comme suit (on suppose que l'on a déjà déclaré la variable `T_abb` comme globale dans l'environnement courant) :

```
->ajout_abonne("Marianne",20,"Paris")
```

ou bien comme suit :

```
->ajout_abonne("Marianne",20)
```

Si la ville n'est pas Paris, on doit utiliser la syntaxe longue. Noter aussi que le type de la variable d'entrée `age` est testé et si elle n'est pas donnée sous forme de chaîne de caractères mais sous forme d'un nombre, elle est convertie. Cette fonction peut être utilisée pour définir une fonction mettant à jour `T_abb` de manière interactive :

```
function t_abb = MiseAJour(t_abb)
  reponse = input("ajouter(a), supprimer(s), quit(q)?",..
                                "string")
  reponse = part(reponse,1);
  if reponse == "q" then return;end
  global T_abb
  T_abb = t_abb
  if reponse == "a" then
    nom=input("Quel nom ?","string")
    age = 0;ierr = 1;
    while   age < 1 | age > 150
      age = input("Quel age ?")
    end
    ville = input("Quelle ville (défaut = Paris)?","string")
    if stripblanks(ville) == emptystr() then
      ajout_abonne(nom,age)
    else
      ajout_abonne(nom,age,ville)
    end
  elseif reponse == "s" then
    nom = input("Quel nom à supprimer?","string")
    supprime_abonne(nom)
  end
  t_abb = T_abb
```

```
  clearglobal T_abb
endfunction
```

Fichier source : `scilab/MiseAJour.sci`

Dans cette fonction on a utilisé la fonction `input` pour dialoguer avec l'utilisateur et acquérir des chaînes (option `"string"`) ou des nombres. La fonction `stripblanks` supprime tous les caractères blancs présents au début et à la fin d'une chaîne de caractères. La fonction `clearglobal` est utilisée pour supprimer la variable globale `T_abb` qui n'est plus utile lorsque la fonction `MiseAJour` se termine.

En ce qui concerne les variables de sortie, on peut aussi appeler une fonction avec moins d'arguments que ceux de la définition de la fonction. Ce nombre effectif d'arguments de sortie peut être utilisé par la fonction pour éviter des calculs inutiles (ou même changer la nature des sorties). Supposons par exemple que l'on veuille avoir une fonction qui donne le profil d'un abonné à partir de son numéro (position dans la liste `T_abb`).

```
function [nom,age,ville]=requete_abonne(numero)
  global T_abb
  lhs=argn(1)
  nom=T_abb(numero,1)
  if lhs > 1 then age=T_abb(numero,2);end
  if lhs > 2 then ville=T_abb(numero,4);end
endfunction
```

Fichier source : `scilab/requete_abonne.sci`

La fonction `argn` appelée comme `argn(1)` retourne seulement le nombre d'arguments de sortie (`lhs`) et bien entendu `argn(2)` retourne seulement le nombre d'arguments d'entrée (`rhs`).

Dans cet exemple les tests sur `lhs` permettent d'éviter les calculs inutiles. Évidemment, dans ce cas particulier, le gain n'est pas significatif.

On a vu que dans un appel de fonction, le nombre d'arguments d'entrée ou de sortie peut être variable, mais pas dans la définition de la fonction. Dans certaines applications, on peut ne pas connaître a priori le nombre d'arguments d'entrée ou de sortie. Pour cela, Scilab permet une construction avec les deux mots clés `varargin` et `varargout`.

Par exemple la fonction `affiche` définie ici peut être appelée avec un nombre arbitraire d'arguments d'entrée. Ces arguments sont alors placés dans `varargin` qui est disponible à l'intérieur de la fonction et qui peut être utilisé comme une structure de type `list` (voir la section 6.1).

```
function affiche(varargin)
  texte=emptystr()
  for x=varargin
```

```
    if typeof(x)~="string" then x=string(x);end
    texte=texte+x
  end
  mprintf("%s\n",texte)
endfunction
```

Fichier source : `scilab/affiche.sci`

Cette fonction affiche ses arguments d'entrée (aussi bien des nombres que des chaînes de caractères) sur une même ligne.

```
->for v=T_abb'
-> affiche(v(1)," est agé(e) de ",v(2),..
                        " ans et habite ",v(3))
->end
Francois est agé(e) de 32 ans et habite Paris
Pierre est agé(e) de 31 ans et habite Paris
Tina est agé(e) de 6 ans et habite Paris
Marianne est agé(e) de 20 ans et habite Paris
```

Rappelons que l'instruction `for v=T_abb'` signifie que la variable `v` prend comme valeurs successives les colonnes de `T_abb'`, c'est-à-dire les lignes de `T_abb`.

3.4.3 Application

On va donner maintenant un exemple beaucoup plus complexe de manipulation des arguments d'entrée-sortie. On cherche un mécanisme permettant de pouvoir redéfinir n'importe quelle fonction de Scilab de façon qu'à chaque fois que cette fonction est appelée, elle affiche ses arguments d'entrée et de sortie. Cet utilitaire pourrait être utile pour le débogage et la surveillance de programmes complexes.

Supposons que la fonction que l'on désire redéfinir s'appelle `pipo`. La nouvelle fonction `pipo` doit d'abord afficher ses entrées, puis faire appel à l'ancienne fonction `pipo` et enfin afficher ses sorties. Il est clair que comme le nouveau `pipo` contient un appel à l'ancien `pipo`, on doit sauvegarder `pipo` dans une autre variable, par exemple `pipo_old_`. Cela peut se faire simplement par l'instruction :
`pipo_old_ = pipo`
car, dans Scilab, une fonction est un objet comme un autre.

Pour pouvoir faire appel à `pipo_old_` à l'intérieur de `pipo`, on a évidemment besoin de connaître le nombre et les valeurs des arguments d'entrée de `pipo`, pour les transmettre à `pipo_old_`. Cela se fait facilement en utilisant `varargin`. Il faut aussi connaître le nombre d'arguments de sortie ; pour cela, on peut utiliser la fonction `argn` vue précédemment.

La fonction `_trace` ci-dessous réalise ce qu'on veut :

```
function _trace(%Fonc_name)
// %Fonc_name est le nom de la fonction
```

```
  // Redéfinition de pipo
  function varargout=%Fonc(varargin)
    // visualisation des arguments d'entrée
    mess="Fonction "+%Fonc_name+": arguments entrée:"
    disp(varargin,mess)
    // appel de l'ancienne fonction
    %out="["+strcat("out" + string(1:argn(1)),",")+"]"
    execstr(%out+"="+%Fonc_name+"_old_(varargin(:))")
    execstr("varargout=list("+%out+")")
    // visualisation des arguments de sortie
    mess="Fonction "+%Fonc_name+": arguments de sortie:"
    disp(varargout,mess)
  endfunction

  // Renvoi du nouveau pipo et pipo_old_ vers l'extérieur
  execstr("["+%Fonc_name+","+%Fonc_name+"_old_] = "+..
"resume(%Fonc,evstr(%Fonc_name))")
endfunction
```

Fichier source : `scilab/_trace.sci`

Ici, `%Fonc_name` est une chaîne de caractères contenant le nom de la fonction à redéfinir. La première étape est de définir la fonction `%Fonc` comme suit :

```
function varargout=%Fonc(varargin)
 <Code>
endfunction
```

Finalement on exécute l'instruction[5] :

```
execstr("["+%Fonc_name+","+%Fonc_name+"_old_] = "+..
                  "resume(%Fonc,evstr(%Fonc_name))")
```

qui termine la fonction en renvoyant dans l'environnement d'appel la fonction `%Fonc` sous le nom `pipo` et la fonction `pipo_old_` sous son propre nom.

Pour comprendre comment marche la fonction `%Fonc`, noter que `argn(1)` donne le nombre d'arguments de sortie effectifs de `pipo`. À partir de ce nombre, on construit une chaîne de caractères `%out` qui contient la liste de sortie `%out1` ,`%out2`,... de longueur `argn(1)`. Après l'affichage du nom de la fonction et des arguments d'entrée, on exécute l'instruction :

```
execstr("["+%out+"]=pipo_old_(varargin(:))")
```

qui fait appel à la fonction `pipo_old_`. Les résultats de cet appel sont renvoyés dans `%out1`,`%out2`,... et mis dans la liste `varargout` par l'instruction :

```
execstr("varargout=list("+%out+")")
```

Enfin le contenu de `varargout` est affiché et on quitte la fonction.

[5] La syntaxe `..` est utilisée pour indiquer qu'une instruction se répartit sur plusieurs lignes.

Appliquons cette méthode de débogage à une petite fonction :

```
 ->function
y=pipo(a,b),y=a+b,endfunction ->_trace("pipo") ->pipo(1,2);
 Fonction pipo: arguments entrée:

        (1)

    1.

        (2)

    2.

 Fonction pipo: arguments de sortie:

        (1)

    3.
```

3.5 Débogage

On a vu comment écrire des fonctions en langage Scilab et comment lancer leur exécution. L'écriture de code s'accompagne inévitablement de bugs. Ce paragraphe présente les différents outils de mise au point disponibles dans Scilab.

Les erreurs pour lesquelles l'interpréteur génère un message (syntaxe, cohérence) sont en général très courantes et faciles à diagnostiquer et à corriger. Dans les cas les plus difficiles, l'usage de la fonction `errcatch` en appel avec l'action `pause` (voir 6.4.3) permet de rendre la main à l'utilisateur au point où l'erreur s'est produite et dans son contexte, ce qui permet d'analyser les variables et de comprendre ce qui se passe. Exemple l'exécution du script :

```
function y=foo(x),
  y=sin(x)/(x^2)
endfunction

errcatch(-1,"pause");
Y=[];X=linspace(-1,1,11);
for k=1:11
  Y=[Y, foo(X(k))];
end
```

Fichier source : `scilab/errcatch.sce`

produit :

```
->exec("errcatch.sce");
 !-error 27
division by zero...
-1->
```

L'utilisateur peut alors analyser la situation :

```
-1->whereami()
whereami called under pause
pause    called at line 3 of macro foo
-1->x,k
 x  =

    0.
 k  =

    6.

-1->abort // terminates the execution
```

Il y a dans Scilab un certain nombre de fonctions qui permettent de suivre l'exécution de programmes et de localiser les erreurs. On va prendre un exemple très simple de fonction :

```
function y=bisq(x,a,b,c)
  xx=x.*x
  y=a*xx.^2+b*xx+c
endfunction
```

Fichier source : `scilab/bisq.sci`

que l'on utilise comme suit après chargement :

```
->  a=1;b=2;c=-4;

->xres=fsolve(100,bisq)
 xres  =

    1.1117859
```

La fonction `fsolve`, que l'on verra dans la section 8.3, permet de résoudre un système d'équations non-linéaires qui peut être défini par une fonction Scilab (ici `bisq` définit l'équation $y = ax^4 + bx^2 + c$).

La première possibilité consiste à interrompre l'exécution d'une fonction écrite en langage Scilab. Cela peut se faire soit par la commande clavier Ctrl-c qui permet d'interrompre l'exécution à l'instant où l'interruption est saisie, mais sans pouvoir préciser à quel endroit du code. Si l'on souhaite spécifier le

point d'interruption, il faut utiliser les fonctions `setbpt` ou `pause`. La commande `pause` doit être insérée dans le code de la fonction (ce qui nécessite un nouveau chargement de la fonction) à l'endroit souhaité alors que la commande `setbpt` (qui place un point d'arrêt ou breakpoint) est une commande extérieure à la fonction qui ne modifie pas son code source.

L'éditeur associé scipad dispose d'un menu spécifique `Debug` qui permet l'insertion de points d'arrêt, l'exécution pas à pas...

```
a=1;b=2;c=-4;x=10;
->bisq(x,a,b,c)
 ans  =

    10196.
->setbpt("bisq",2)

->bisq(x,a,b,c)
Stop after row     2 in function bisq :


-1->
```

La commande `setbpt("bisq",2)` a eu le même effet que l'insertion de la commande `pause` après la deuxième ligne de la fonction `bisq`.

Le prompt `-1->` signifie que nous sommes maintenant au niveau 1 d'interruption (on peut avoir plusieurs niveaux d'interruptions). À ce niveau, on peut faire toutes les manipulations souhaitées sans affecter l'espace des variables (section 3.4.2) du niveau appelant. Ce qui est essentiel c'est que l'on a accès « en lecture » à toutes les variables des niveaux inférieurs. Il est par exemple possible d'observer la taille et la valeur des variables :

```
 -1->size(xx)
 ans  =

!   1.    1. !

-1->xx
 xx  =

    100.
```

S'il n'est pas possible de modifier la valeur d'une variable contenue dans l'espace des variables du niveau appelant, il est possible d'en créer une copie locale qui sera modifiable. La manière la plus simple consiste à taper l'instruction `<nom>=<nom>;` ou, de façon équivalente, tout simplement l'instruction `<nom>;`. Ici `<nom>` désigne le nom de la variable.

Oublier de dupliquer une variable dont on souhaite modifier une composante est une erreur courante que nous allons illustrer de manière plus explicite par un exemple très simple :

```
->a=[1 2 3 4];
->pause;
-1->a(2)=3.14
 a  =

!   0.   !
!   3.14 !
-1->clear a;//on efface la variable locale
-1->a; //on crée une copie locale
-1->a(2)=3.14 //on modifie la copie locale
 a  =

!   1.    3.14    3.    4. !
```

On redescend au niveau inférieur et on reprend l'exécution interrompue par la commande `resume` ou `return`.

Il est possible de modifier des variables du niveau inférieur à partir du niveau d'interruption par la commande `resume` en lui donnant comme paramètre les noms des variables modifiées. Dans le cas de la fonction `bisq` précédente on peut faire :

```
->y=bisq(x,a,b,c)
Stop after row     2 in function bisq :


-1->a=5;b=0.;

-1->[a,b]=resume(a,b)
 y  =

    49996.
```

On peut décider d'abandonner l'exécution et de revenir directement au niveau 0 par la commande `abort`.

Les points d'interruptions créés par la commande `setbpt` sont listés par la commande `dispbpt` et libérés par la commande `delbpt` :

```
->dispbpt()
 breakpoints of  function :bisq

         2

->delbpt("bisq",2)
```

Lorsque l'on utilise Ctrl-c pour interrompre l'exécution de longs programmes ou lorsque que l'on a positionné plusieurs points d'interruption, il est utile de localiser l'endroit où l'interruption s'est produite. Cela est réalisé par la commande `whereami` :

```
-1->whereami()
whereami called under pause
pause    called at line 3 of macro bisq
```

ou la commande `where` :

```
-1->[linenum,fun]=where()
 fun  =

!pause  !
!       !
!bisq   !
 linenum  =

!   0. !
!   3. !
```

`fun` donne la hiérarchie des fonctions appelantes et `linenum` les numéros de lignes correspondantes des codes source. Si la fonction est une fonction des bibliothèques Scilab (voir la section 6.3) on peut localiser le fichier source de cette fonction en utilisant la fonction `get_function_path` :

```
->get_function_path("median")
 ans  =

 /usr/local/lib/scilab/macros/elem/median.sci
```

Nous avons vu qu'une fonction peut également être exécutée par `exec` ; cette commande est très utile pour la correction d'erreurs car elle permet une exécution pas à pas de la fonction et l'affichage des résultats intermédiaires. Bien entendu, dans ce cas, il faut au préalable définir les arguments d'entrée.

Reprenons la fonction `bisq` ci-dessus :

```
->a=1;b=2;c=-4;x=10;

->exec(bisq,7)
step-by-step mode: enter carriage return to proceed
»
 xx  =

    100.
»
 y  =

    10196.
```

```
»

->
```

Après chaque instruction Scilab affiche l'invite >> et attend que l'utilisateur saisisse un retour chariot ou déclenche une pause. À la fin de l'exécution les variables locales ne sont pas effacées.

Si la fonction a été chargée avec l'option `"n[ocompile]"` de `getf`[6] :

```
getf("bisq.sci","n")
```

alors l'exécution affiche aussi le texte des instructions :

```
->a=1;b=2;c=-4;x=10;

->exec(bisq,7)
step-by-step mode: enter carriage return to proceed
»
xx=x.*x
 xx  =

    100.
»
y=a*xx.^2+b*xx+c
 y  =

    10196.
»
return!

->
```

Les erreurs les plus difficiles à trouver sont celles qui dépendent de l'environnement, c'est-à-dire qui ne se produisent que pour certaines valeurs des paramètres. Or, pour corriger une erreur, il faut pouvoir la reproduire : pour cela il est commode d'utiliser les fonctions `save` et `load`. En utilisation courante, ces fonctions permettent de sauver et de recharger des variables de manière efficace (en format binaire indépendant du système hôte). L'appel le plus simple `save("nom de fichier")` permet de sauver toutes les variables utilisées pour recharger ensuite cet environnement et reproduire une erreur éventuelle.

Pour les problèmes vraiment récalcitrants, il reste encore les commandes `clearfun` et `debug` qui nécessitent la connaissance du fonctionnement interne de Scilab.

[6] Par défaut, la fonction `getf` exécute une pré-interprétation (compilation) du code de la fonction et génère un pseudo-code opératoire plus efficace. L'option `"n"` supprime cette pré-interprétation.

Chapitre 4

Fonctions d'entrée-sortie

4.1 Introduction

La plupart des programmes ont besoin d'acquérir, de stocker ou d'afficher des données. Dans le cas des programmes de calcul numérique, il s'agit essentiellement de pouvoir lire et écrire des nombres entiers ou flottants et des chaînes de caractères.

Les nombres peuvent être représentés directement par leur codage interne en séquence de bits, on parlera alors d'entrée-sortie binaire. Ils peuvent aussi être représentés comme du texte ASCII. Dans ce cas la représentation interne est convertie en une forme lisible par l'utilisateur, on parlera alors d'entrée-sortie formatée. Notons que pour les chaînes de caractères, il n'y a pas de différence entre ces deux types de représentations.

Les entrées-sorties formatées présentent l'avantage d'être facilement lues ou écrites par l'utilisateur : il est en effet plus facile d'entrer le seul caractère « 2 » pour désigner l'entier 2 que de saisir la séquence de 32 bits :

00000000000000000000000000000010

correspondant à sa représentation machine. Bien évidemment le formatage induit un surcoût de calcul. Les entrées-sorties binaires permettent également de donner la représentation des nombres sans erreur d'arrondi par rapport à leur représentation interne.

4.2 Dialogue avec l'utilisateur

Scilab offre plusieurs modes de dialogue avec l'utilisateur. Cela peut se faire par l'intermédiaire de l'interface graphique décrite en section 4.6 ou directement sous forme textuelle dans la fenêtre d'entrée-sortie de Scilab.

4.2.1 Visualisation textuelle standard

Lorsque l'utilisateur fait exécuter une instruction Scilab terminée par une virgule ou un retour à la ligne, le logiciel produit automatiquement une visualisation du résultat en utilisant un formatage spécifique au type du résultat :

```
->A=[1/3 2;3 4.5-0.55*%i]
 A  =

!   0.3333333    2.          !
!   3.           4.5 - 0.55i !


->P=3*%s^4-2*%s
 P  =

           4
  - 2s + 3s
```

Cette visualisation automatique est désactivée lorsque l'instruction se termine par un point-virgule ou lorsqu'elle se trouve à l'intérieur d'une fonction Scilab. Mais le programmeur peut alors explicitement demander la visualisation d'une ou plusieurs variables en utilisant la fonction `disp` :

```
->A = [1 2;0 4];
->B = [1;1];
->X = A\B;
->disp(X,"La solution de l""équation A*X = B vaut:")
 La solution de l'équation A*X = B vaut

!   0.5  !
!   0.25 !
->disp(union) //union est une fonction

[x,ka,kb] = union(a,b)
```

Contrairement à la visualisation en mode interactif, le nom de la variable `X` n'est pas affiché. La fonction `print` permet d'obtenir l'affichage correspondant au mode interactif. Son premier argument désigne le nom du fichier ou l'unité logique vers lequel on dirige l'affichage.

```
->H = (%s^2-1)/(3*%s^4-2*%s);

->print(%io(2),H);
 H  =

          2
```

```
   - 1 + s
    ----
           4
  - 2s + 3s
```

L'expression `%io(2)` indique l'unité logique de sortie (voir la section 4.3.2) associée à la fenêtre d'entrée-sortie Scilab, alors que `%io(1)` indique l'unité logique d'entrée associée à la fenêtre Scilab.

Le format d'affichage des nombres peut être spécifié en utilisant la fonction `format`. Il est ainsi possible de fixer le nombre maximum de caractères pouvant être utilisés pour représenter un nombre :

```
->[1/3, 2.5]
 ans  =

!   0.3333333    2.5 !
->format(15);
->[1/3, 2.5]
 ans  =

!   0.333333333333    2.5 !
```

Ces changements de format n'affectent évidemment pas la précision des calculs, seule la visualisation change !

Lorsqu'une valeur d'un tableau est négligeable par rapport à la plus grande valeur du tableau elle est affichée comme `0` :

```
 ->x = [1000, 1, 1.d-13]
 x  =

!   1000.    1.    0. !
->x(3)
 ans  =

    1.000D-13
```

Par défaut, les affichages sont réalisés avec un format variable qui utilise la représentation la mieux adaptée pour chaque nombre à représenter. Il est aussi possible de demander un affichage au format scientifique :

```
->format("e",10);
->[1/3, 2.5]
 ans  =

!   3.333D-01    2.500D+00 !
->format(13)
->[1/3, 2.5]
 ans  =

!   3.333333D-01    2.500000D+00 !
->format("v",10) //retour au format par défaut
```

4.2.2 Visualisation textuelle avancée

L'utilisateur peut souhaiter afficher des résultats avec un format particulier. Il est alors possible d'utiliser la fonction `mprintf` qui émule la procédure `printf` du langage C.

```
->n_iter = 33;
->alpha = 0.535;
->mprintf("Itération: %i, Résultat: alpha=%f\n", n_iter,alpha)
Itération: 33, Résultat: alpha=0.535000
```

On notera ici l'utilisation des formats « C » `%i` pour afficher des entiers et `%f` pour afficher des nombres flottants ainsi que `\n` qui dénote le retour à la ligne. Cette fonction est très puissante mais requiert la connaissance des formats C. Une solution alternative peut être d'utiliser les fonctions et opérations de manipulation de chaînes de caractères fournies dans Scilab.

```
->n_iter = 33;
->alpha = 0.535;
->disp("Itération numéro: "+string(n_iter)+..
             ", Le résultat vaut: alpha="+string(alpha))

 Itération numéro: 33, Le résultat vaut: alpha=0.535
```

Dans ce cas c'est la fonction `string` qui se charge du formatage des nombres, en utilisant la même procédure que celle utilisée pour les fonctions `disp` ou `print`.

La fonction `mprintf` permet aussi de visualiser des tableaux de nombres et de chaînes :

```
->colors = ["rouge";"vert";"bleu";"rose";"noir"];
->RGB = [1 0 0;0 1 0;0 0 1;1 0.75 0.75;0 0 0];
->mprintf("\nNoms\tR        \tG        \tB\n");
->mprintf("%s\t%f\t%f\t%f\n",colors,RGB);

Noms    R               G               B
rouge   1.000000        0.000000        0.000000
vert    0.000000        1.000000        0.000000
bleu    0.000000        0.000000        1.000000
rose    1.000000        0.750000        0.750000
noir    0.000000        0.000000        0.000000
```

L'exemple ci-dessus montre que la fonction `mprintf` réitère le format pour chaque ligne des tableaux de données. Il montre aussi l'utilisation du format `\t` qui produit une tabulation ainsi que du format `%s` qui permet l'affichage d'une chaîne de caractères.

4.2.3 Lecture de données à l'écran

Dans certains cas, un programme en cours d'exécution peut avoir à demander à l'utilisateur de fournir des données supplémentaires. La fonction `input` envoie un message à l'utilisateur et se met en attente d'une réponse qui peut être un simple nombre, une séquence de nombres ou une chaîne de caractères selon l'option spécifiée.

Si dans le code du programme le programmeur a écrit l'instruction suivante :

```
n_iter = input(["L""algorithme n""a pas convergé,";
    "donnez le nombre d""itérations supplémentaires"]);
```

l'utilisateur obtiendra le message :

```
L'algorithme n'a pas convergé
donnez le nombre d'itérations supplémentaires ->
```

et il pourra alors donner la valeur à la suite du prompt `-->`.

Donnons un autre exemple de programme qui demande à l'utilisateur de fournir les caractéristiques d'un individu pour créer une base de données :

```
name = input("Donnez le nom d'une personne","string");
while %t
  data = input("Donnez son age, sa taille et son poids");
  if size(data,"*") == 3 then break,end
  mprintf("Saisie incorrecte !\n");
end
mprintf("\nOk: %s, age=%d, taille=%.2fm, poids=%.1fkg\n",..
                          name, data(1), data(2), data(3));
```

Fichier source : `scilab/io1.sce`

Ici le premier appel à la fonction `input` avec l'option `"string"` attend la saisie d'une chaîne de caractères et les suivants, dans la boucle infinie `while %t`, attendent un vecteur de nombres. Le dialogue utilisateur sera alors :

```
Donnez le nom d'une personne->Serge
Donnez son age, sa taille et son poids->47 1.68 85

Ok: Serge, age=47, taille=1.68m, poids=85.0kg
```

On notera dans cet exemple l'utilisation du format `%.nf` qui précise le format `%f` en fixant le nombre `n` de décimales affichées.

Le code précédent peut aussi être écrit en utilisant la fonction `mscanf`, émulation de la fonction C `scanf` :

```
mprintf("Donnez le nom d'une personne,"+..
            " son age, sa taille et son poids\n");
while %t
```

```
  [n,name,age,taille,poids] = mscanf("%s %d %f %f")
  if n == 4 then break,end
  //l'utilisateur n'a pas fourni 4 valeurs
  mprintf("Saisie incorrecte !\n");
end
mprintf("\nOk: %s, age=%d, taille=%.2fm, poids=%.1fkg\n",..
                                  name, age, taille, poids)
```

Fichier source : `scilab/io2.sce`

et l'on obtient alors le dialogue utilisateur suivant :

```
Donnez le nom d'une personne, son age, sa taille et son poids
->Serge 47 1.68
Saisie incorrecte !
->Serge 47 1.68 85
Ok: Serge, age=47, taille=1.68m, poids=85.0kg
```

Dans certains cas, le programmeur souhaite que le programme demande à l'utilisateur le droit de pouvoir continuer l'exécution, par exemple pour lui laisser le temps de voir ce qui a été écrit à l'écran. Cela peut se faire simplement par l'instruction `halt()`. L'utilisateur reçoit alors le message d'invite : `halt-->` et l'exécution reprend dès que l'utilisateur frappe un retour à la ligne.

4.3 Lecture et écriture des fichiers

Dès que le nombre de données nécessaires à l'exécution d'un programme dépasse quelques unités, il devient essentiel de pouvoir les lire à partir de fichiers. L'utilisation de fichiers pour les entrées-sorties est cependant un peu plus compliquée que les entrées-sorties à l'écran.

4.3.1 Répertoires et chemins de fichiers

Dans Scilab les fichiers sont désignés par leur chemin, de la même façon que pour le système hôte. Ces chemins sont en général donnés par des chaînes de caractères. Sous Unix/Linux cela peut-être :

```
"/tmp/mon_fichier.sce"
"moteur/mes_donnees.dat"
```

et sous Windows

```
"c:\scilab\mon_fichier.sce"
"moteur\mes_donnees.dat"
```

Ces chemins peuvent être absolus ou relatifs au répertoire courant. Les chemins absolus commencent sous les systèmes Unix par un « / » alors que sous Windows ils commencent le plus souvent par une lettre, identifiant le disque, suivie de « : ».

La plupart des fonctions acceptant des chemins en argument acceptent indifféremment la notation Unix ou Windows et effectuent automatiquement la conversion dans le format adapté au système hôte. La fonction `pathconvert` permet de convertir explicitement un chemin relatif dans la forme adaptée au système hôte. Les fonctions `getlongpathname` et `getshortpathname` permettent de gérer les deux représentations des chemins de Windows. Sous Unix ces deux fonctions n'effectuent aucune modification. L'ensemble des ces fonctions permet d'écrire des codes portables.

Au démarrage de Scilab, le répertoire courant est celui d'où Scilab a été lancé. À chaque instant il est possible de connaître ce répertoire courant en utilisant les commandes `pwd` ou `getcwd`. Il est possible de changer le répertoire courant avec la commande `chdir` ou `cd`.

Scilab définit aussi des abréviations pour trois autres répertoires particulièrement utiles. Le répertoire principal de l'utilisateur (son « home directory ») peut être abrégé en `~/` ; sa définition est contenue dans la variable `home`. Le répertoire d'installation de Scilab peut être abrégé en `SCI/` ; sa définition est contenue dans la variable `SCI`. Enfin, le répertoire temporaire créé au lancement de Scilab et détruit lorsque l'on quitte le logiciel ; sa définition est contenue dans la variable `TMPDIR`.

4.3.2 Ouverture et fermeture des fichiers

Tout d'abord, avant de pouvoir accéder aux données d'un fichier, il faut créer un lien entre le fichier et Scilab. Cela se fait par l'intermédiaire de la fonction `mopen` qui, étant donné une chaîne de caractères décrivant le chemin du fichier et quelques options, retourne un nombre, appelé identificateur ou unité logique de ce fichier, et définit le positionnement initial dans ce fichier[1]. L'identificateur désigne en fait une structure de données interne contenant les informations sur ce fichier (type, position courante, mémoire tampon...).

La fonction `mopen` interface la fonction C `fopen` ; sa syntaxe est :

```
[fd,err]=mopen(fil,mode,swap)
```

`fil` est une chaîne de caractères contenant le chemin du fichier. Par exemple sous Windows

```
"C:Mes Documents\Scilab\dat01"
```

ou sous Unix :

```
"~/Scilab/dat01"
```

Le paramètre `mode` est une séquence de caractères permettant de spécifier si le fichier doit être ouvert en lecture seule (`"r"`), en écriture seule (`"w"`), en complétion (écriture à la fin `"a"`) ou en lecture/écriture (`"r+"`). L'ajout du caractère `"b"` spécifie que les données numériques stockées dans le fichier sont représentées par leur code binaire. En l'absence de cette option, sous Windows,

[1] Certaines fonctions de lecture ou écriture sur les fichiers créent ce lien automatiquement à partir du nom de fichier.

les fichier sont censés être des fichiers textes et le caractère de code ASCII 13 est alors interprété comme un saut de ligne.

La valeur par défaut de `mode` est (`"rb"`). Par défaut, Scilab suppose que la représentation des nombres dans les fichiers binaires correspond au format IEEE *little endian* [2], la conversion au format interne du processeur étant alors faite automatiquement.

L'option `swap=0` permet de désactiver cette conversion, par exemple dans le cas où le fichier à lire contient des nombres au format *big endian*.

La fonction `mopen` retourne l'identificateur du fichier `fd` et éventuellement un indicateur d'erreur.

```
->fd1=mopen("SCI/scilab.star","r")
 fd1  =

    2.
->[fd2,err]=mopen(TMPDIR+"/out","wb")
 err  =

    0.
 fd2  =

    3.


->dispfiles()
|File name                        |Unit|Type|Options       |
|-----------------------------|
|/usr/local/lib/scilab/scilab.star|1   |C   |r             |
|/tmp/SD_20388_/out               |2   |C   |w b           |
|Input                            |5   |F77 |old formatted |
|Output                           |6   |F77 |new formatted |
```

La fonction `dispfiles` permet d'examiner les fichiers ouverts. On remarque ici que les fichiers peuvent être de deux types C (`C`) ou FORTRAN (`F77`). Ce dernier type existe pour permettre de manipuler les fichiers créés par ou pour d'autres logiciels écrits en FORTRAN. La manipulation de ces fichiers est décrite à la page 71. Pour une manipulation avancée des fichiers, la fonction Scilab `file` permet d'acquérir les chemins des fichiers ouverts et leurs propriétés.

```
->[unit,typ,nams,mod,swap]=file()
 swap  =

    0.    0.    0.    0.
 mod  =
```

[2] Le format *little endian* correspond au mode de stockage où le bit de poids faible est stocké en premier contrairement au stockage *big endian*où c'est le bit de poids fort. Cette appellation provient du *Voyage de Gulliver* de Jonathan Swift où les Big-Endians et les Little-Endians discutent pour savoir si les œufs à la coque doivent être ouverts par le petit ou le gros bout...

```
    100.    201.    1.    0.
 nams  =

!/usr/local/lib/scilab/scilab.star  /tmp/SD_20388_/out      !
 typ  =

!C  C  F  F  !
 unit  =

    1.    2.    5.    6.
```

Le nombre de fichiers pouvant être ouverts simultanément est limité, aussi faut-il penser à fermer les fichiers qui ne sont plus utiles. Cela peut être fait par la fonction `mclose` qui doit être utilisée à la fin du traitement du fichier pour déconnecter le fichier de l'environnement Scilab. Si `fd` est une unité logique associée à un fichier, `mclose(fd)` ferme ce fichier. Il faut noter que les fichiers ouverts ne sont pas fermés automatiquement lorsqu'une erreur survient à l'exécution d'une instruction Scilab.

L'instruction `mclose("all")` permet de fermer tous les fichiers ouverts.

4.3.3 Lecture binaire

La lecture et l'écriture de fichiers binaires de type `C` permet d'échanger des données avec la plupart des logiciels actuels. Les fichiers de sons, au format `wav` ou `au`, les fichiers d'images `gif`, `ppm` ou `jpeg`, les fichiers de sauvegarde de Scilab (voir page 73), voire même les fichiers Excel `xls` sont reconnus par Scilab. Ces fichiers sont constitués d'une séquence d'octets, il est donc impératif d'en connaître la structure.

Un exemple valant mieux que mille discours, montrons comment on peut écrire en Scilab la fonction `readppm` qui lit les fichiers d'images de la famille `PPM` (couleur), `PGM` (niveau de gris) et `PBM` (noir et blanc).

Ce type de fichier d'image est structuré en quatre parties :

1. une ligne de texte donnant le type (couleur, niveau de gris ou noir et blanc) et le mode de stockage (ASCII ou binaire) ;
2. des lignes de commentaires débutant par le caractère `#` ;
3. une ligne contenant les dimensions de l'image en pixels et selon les cas une ligne contenant le nombre de couleurs ;
4. la séquence des nombres caractérisant la couleur de chacun des pixels selon le type d'image considéré.

Donnons tout d'abord la fonction permettant d'ouvrir le fichier et de tester s'il correspond au type désiré.

```
function [ftyp,fd]=open_ppm(fil)
  [fd,err]=mopen(fil,"rb")
```

```
  if err<>0 then error("Le fichier "+fil+" n'existe pas");end

  // lecture et test du type qui doit être
  //P1, P4 : image noir et blanc
  //P2, P5 : image en niveau de gris
  //P3, P6 : image couleur
  h=mgetl(fd,1)
  ftyp=find(h=="P"+string(1:6))
  if ftyp==[] then
    error("Ce n'est pas un fichier de type PBM/PGM/PPM");
  end
endfunction
```

Fichier source : `scilab/open_ppm.sci`

On retrouve ici l'utilisation de la fonction `mopen` et l'on découvre celui de la fonction `mgetl` qui permet de lire une ou plusieurs lignes du fichier comme du texte. Les lignes de texte sont supposées terminées par des marques de fin de ligne : (« `\r\n` ») sous Windows, (« `\r` ») sous Macintosh et (« `\n` ») pour les fichiers Unix. « `\r` » correspond au code ASCII 13 et « `\n` » correspond au code ASCII 10. La fonction `mgetl` gère ces différentes terminaisons de ligne indépendamment de la plateforme. Pour lire non pas une ligne entière mais une chaîne de caractères de longueur donnée, on aurait utilisé la fonction `mgetstr`.

La fonction `ppm_get_comments` ci-dessous extrait les commentaires du fichier.

```
function head=ppm_get_comments(fd)
  head=[];
  while %t
    pos=mtell(fd);//mémorise la position courante
    ligne=mgetl(fd,1) //lecture de la ligne suivante
    first=part(ligne,1) //le premier caractère de la ligne
    if first ~="#" then
      mseek(pos,fd) //positionnement au point mémorisé
      break
    end
    head=[head;ligne]
  end
endfunction
```

Fichier source : `scilab/ppm_get_comments.sci`

Les fonctions `mtell` et `mseek` utilisées ci-dessus permettent respectivement de mémoriser la position courante dans un fichier et de la (re-)positionner.

Ensuite la fonction `ppm_get_size` permet de lire les dimensions de l'image et, s'il existe, le nombre de couleurs :

```
function [sz,nc]=ppm_get_size(fd,ftyp)
  // Dimensions de l'image
  ligne=mgetl(fd,1);
  execstr("sz=["+ligne+"]");
  // Nombre de couleurs
  if and(ftyp<>[1,4]) then
    ligne=mgetl(fd,1);
    execstr("nc="+ligne);
  else
    nc=[];
  end
endfunction
```

Fichier source : `scilab/ppm_get_size.sci`

On observe ici l'utilisation de la fonction `execstr` qui permet d'évaluer une instruction Scilab donnée par une chaîne de caractères.

Enfin la fonction `ppm_get_bin` permet de lire la valeur des pixels dans le cas d'un stockage en binaire :

```
function img=ppm_get_bin(fd,ftyp,sz)
  select ftyp
  case 4 then //blanc et noir 8 pixels codés dans un caractère
    t=mgeti((sz(1)*sz(2))/8,'uc',fd);
    f=uint8(2.^(0:7))
    img=matrix([(t&f(8))/f(8);(t&f(7))/f(7);
                (t&f(6))/f(6);(t&f(5))/f(5);
                (t&f(4))/f(4);(t&f(3))/f(3);
                (t&f(2))/f(2);(t&f(1))/f(1)],sz);
  case 5 then // niveau de gris- un octet par pixel
    img=matrix(mgeti(sz(1)*sz(2),'uc',fd),sz)
  case 6 then // couleur - 3 octets par pixels
    img=matrix(mgeti(3*sz(1)*sz(2),'uc',fd),[3,sz])
  end
endfunction
```

Fichier source : `scilab/ppm_get_bin.sci`

Dans le cas « noir et blanc » les pixels sont stockés à raison de 8 pixels par octet. L'opérateur `&` appliqué à des variables de type entier effectue le « ET » bit à bit de ses deux opérandes. On notera la fonction `uint8` qui convertit une variable réelle en une variable contenant des entiers codés sur un octet sans bit de signe.

La fonction `mgeti` permet de lire dans un fichier binaire des nombres entiers codés sur 1, 2 ou 4 octets et de retourner un vecteur d'entiers de type correspondant.

La fonction `mget`, dont la syntaxe et le fonctionnement ressemblent à ceux de `mgeti`, permet de lire des données flottantes ou entières et de les convertir automatiquement en un vecteur Scilab de réels.

Le premier argument de ces fonctions donne le nombre d'entiers à lire, le second le type des données que l'on souhaite lire :

Type entier	Code	Type entier	Code
char (un octet)	"c"	unsigned char	"uc"
short (deux octets)	"s"	unsigned short	"us"
int (4 octets)	"i"	unsigned int	"ui"
float (4 octets)	"f"	double (8 octets)	"d"

Ces codes peuvent être complétés par un caractère supplémentaire « l » ou « b » pour forcer l'interprétation « little endian » ou « big endian ».

La fonction `matrix` permet de transformer un vecteur (le premier argument) en un tableau dont les dimensions sont données par le dernier argument, les éléments du vecteur formant les colonnes successives du tableau résultat. Dans le cas « couleur », remarquons que `matrix` retourne un tableau à trois dimensions.

La visualisation graphique des images codées en niveau de gris (PGM) est réalisée simplement par la fonction :

```
function gray_image(img)
  [M,N]=size(img)
  mx=double(max(img))
  f=gcf(); //handle sur la figure courante
  f.color_map=graycolormap(mx+1);
  f.axes_size=[M,N];
  a=gca(); //handle sur le changement de coordonnées
  a.margins=[0 0 0 0];//suppression des marges
  Matplot(m',"020") //dessin de l'image
endfunction
```

Fichier source : `scilab/gray_image.sci`

On remarque ci-dessus l'utilisation de la fonction `double`, qui force la conversion des entiers en nombres flottants habituels, ainsi que le recours aux fonctions graphiques (voir chapitre 5). La fonction `graycolormap` permet ici de choisir une table des couleurs en niveaux de gris adaptée à l'image et la fonction `Matplot` réalise l'affichage proprement dit. L'utilisation d'objets de type `handle` pour désigner les objets graphiques et accéder à leurs propriétés sera décrite au chapitre 5.

4.3.4 Lecture formatée

Considérons maintenant l'exemple de la lecture des fichiers PPM ASCII pour illustrer la lecture des données formatées. L'en-tête des fichiers est le même que précédemment. La partie contenant la valeur des pixels est dans ce cas une séquence d'entiers formatés sur 1 ou 3 caractères et séparés par un caractère blanc. La fonction de lecture s'écrit alors très simplement.

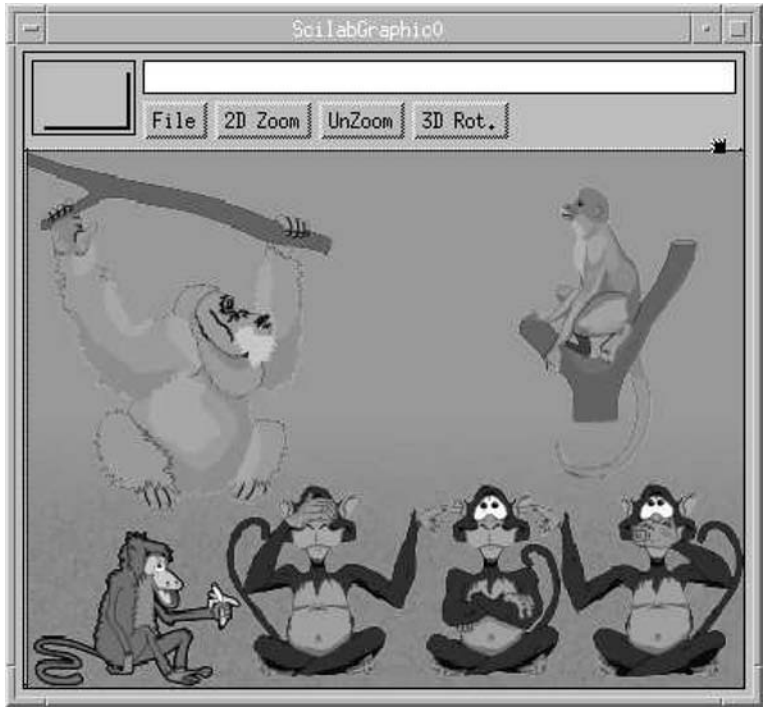


Figure 4.1 – Une image PGM (l'équipe Scilab au travail).

```
function img=ppm_get_ascii(fd,ftyp,sz)
  //Lit les valeurs des pixels pour une image ASCII
  if or(ftyp==[4 5]) then //N&B ou niveau de gris
    img=matrix(uint8(mfscanf(sz(1)*sz(2),fd,"%i ")),sz)
  else
    img=matrix(uint8(mfscanf(3*sz(1)*sz(2),fd,"%i ")),[3,sz])
  end
endfunction
```

Fichier source : `scilab/ppm_get_ascii.sci`

Cet exemple montre une utilisation simple de la fonction `mfscanf` fondée sur la procédure C `fscanf`. Cette fonction a un comportement comparable à la fonction `mscanf` vue précédemment. Dans cet exemple, il est nécessaire de lire une grande quantité de données ; le recours à une boucle interprétée pour effectuer cette lecture ne serait pas efficace. Cette boucle est réalisée par la fonction elle même. Le premier argument de la fonction `mfscanf` indique le nombre de lectures à effectuer au format de lecture `%i`.

Dans cet exemple les données à lire sont toutes de même type mais il est évidemment possible de lire des fichiers contenant des données hétérogènes.

Soit le fichier `ville.dat` :

```
Paris 2125246 Paris
Rocquencourt 3871 Yvelines
Versailles 87789 Yvelines
Le_Mans 145502 Sarthe
...
```

Chaque ligne contient le nom d'une ville, son nombre d'habitants et son département ; la séquence d'instructions :

```
fd=mopen("ville.dat","r");
[n,ville,habitants,departement]=mfscanf(-1,fd,"%s %i %s");
mclose(fd)
```

permet de lire l'ensemble des données du fichier. La valeur `-1` passée comme premier argument de `mfscanf` indique que l'on souhaite lire toutes les lignes du fichier.

Il est important de noter que la ville « Le Mans » a du être écrite `Le_Mans` pour être comprise comme une seule entité par le format `%s`. Pour lire du texte contenant des blancs il faut utiliser le format `%c` à condition bien entendu de connaître le nombre de caractères à lire. Ainsi, le fichier `CodePostal.dat` ci-dessous :

```
 Agen                              47000
 Aigues Juntes                     09000
 Aiguilhe                          43000
 Ainac                             04000
 Ajaccio                           20000
 Ajoux                             07000
 Albi                              81000
 Alencon                           61000
...
```

pourra être lu comme suit (le premier argument de `mfscanf` positionné à `-1` indique que l'on souhaite lire toutes les lignes jusqu'à la fin du fichier) :

```
fd=mopen("CodePostal.dat","r");
[communes,codes]=mfscanf(-1,"%31c %i ")
mclose(fd)
```

ou encore commune par commune

```
fd=mopen("CodePostal.dat","r");
while meof()==0
  [commune,code]=mfscanf("%31c %i)
end
mclose(fd)
```

La fonction `meof` permet de tester si la fin de fichier a été atteinte.

4.3.5 Écriture binaire et formatée

Les fonctions permettant d'écrire des données dans un fichier binaire sont similaires à celles utilisées pour la lecture, les fonctions `mgetl`, `mget`, `mgetstr` étant remplacées par les fonctions `mputl`, `mput`, `mputstr`. Pour l'écriture formatée, la fonction `mfprintf` remplace la fonction `mfscanf`.

Le script ci-dessous peut par exemple être utilisé pour générer le fichier utilisé en page 70

```
villes       = ["Paris";"Rocquencourt";"Versailles";"Le_Mans"];
habitants    = [2125246; 3871         ; 87789      ; 145502  ];
departements = ["Paris";"Yvelines";    "Yvelines";  "Sarthe"];
fd=mopen("ville.dat","w");
mfprintf(fd,"%s %i %s\n",villes,habitants,departements);
mclose(fd)
```

4.4 Entrée-sortie FORTRAN

L'intérêt essentiel des fonctions d'entrée-sortie FORTRAN dans Scilab est de permettre la lecture et la génération de fichiers pour communiquer avec d'autres logiciels du monde FORTRAN. Elles peuvent aussi être utiles aux

utilisateurs qui ont l'habitude de programmer avec ce langage. C'est pourquoi nous ne décrirons que les aspects qui nous paraissent essentiels.

À la différence de C, les entrées-sorties FORTRAN sont structurées en lignes ou en «enregistrements». Les unités logiques utilisées par les fonctions d'entrée-sortie FORTRAN ne sont pas les mêmes que pour les identificateurs C. Pour les fonctions du monde FORTRAN, les fichiers doivent être ouverts par l'instruction `file("open",...)` et fermés par l'instruction `file("close",...)`

4.4.1 Entrée-sortie formatée

Les entrées formatées sont réalisées par la fonction `read` et les sorties formatées par la fonction `write`. Le fichier `CodePostal.dat`, défini page 71, pourrait être lu comme suit :

```
function  [communes,codes]=ReadCP()
  u=file("open","CodePostal.dat","old","formatted")
  communes=read(u,-1,1,"(a31)");
  file("rewind",u) //repositionnement en début de fichier
  codes=read(u,-1,1,"(32x,f5.0)");
  file("close",u)
endfunction
```

Fichier source : `scilab/ReadCP.sci`

Le dernier argument de la fonction `read` donne le format FORTRAN utilisé : `a` pour les chaînes et `f` ou `d` pour les nombres. Le format `32x` indique qu'il faut sauter 32 caractères. Scilab ne permet pas le mélange des types dans les formats, donc, si l'on souhaitait générer le fichier `CodePostal.dat` à partir des tableaux `communes` et `codes`, il faudrait procéder comme suit :

```
function WriteCP(communes,code)
  communes=part(communes,1:32)
  code=string(code)
  write("CodePostal.dat",communes+codes)
endfunction
```

Fichier source : `scilab/WriteCP.sci`

La fonction `part`, dont on a déjà vu l'utilisation pour extraire des sous-chaînes page 15, permet ici de générer des chaînes de longueurs `32` (au besoin complétées par des caractères blancs).

Dans le cas où l'on passe directement le nom du fichier à la fonction `write` (ou `read`), le fichier est automatiquement ouvert puis refermé après écriture (ou lecture).

4.4.2 Entrée-sortie binaire

Les fonctions d'entrée-sortie binaire de type FORTRAN sont `readb` et `writeb` pour les données flottantes double précision (8 octets) et `read4b` et `write4b` pour les données flottantes sur 4 octets. Le code suivant permet par exemple de stocker une matrice dans un fichier binaire FORTRAN :

```
function F77WriteMat(A,fichier)
  u=file("open",fichier,"unformatted")
  write4b(u,size(A))
  writb(u,A)
  file("close",u)
endfunction
```

Fichier source : `scilab/F77WriteMat.sci`

L'argument `"unformatted"` de la fonction `file` spécifie un fichier binaire. L'autre valeur possible de cet argument est `"formatted"` qui est la valeur par défaut. La fonction suivante permet de relire les fichiers créés par la fonction `F77WriteMat` ci-dessus.

```
function A=F77ReadMat(fichier)
  u=file("open",fichier,"unformatted","old")
  sz=read4b(u,1,2)
  A=matrix(readb(u,sz(1),sz(2))
  file("close",u)
endfunction
```

Fichier source : `scilab/F77ReadMat.sci`

L'argument `"old"` de la fonction `file` spécifie que le fichier doit préexister. Les autres valeurs possibles de cet argument sont `"new"` et `"unknown"`.

4.5 Entrées-sorties spécialisées

Au delà des fonctions d'intérêt général qui ont été présentées dans ce paragraphe, Scilab gère des fonctions d'entrée-sortie spécialisées.

4.5.1 Fonctions de sauvegarde des variables Scilab

La fonction `save` permet de sauver une ou plusieurs variables Scilab, sans restriction de type, dans un fichier binaire de structure spécifique. Si l'on ne spécifie que le nom du fichier à créer (`save("fichier")`) alors toutes les variables définies sont sauvegardées. L'exemple ci-dessous montre comment préciser les variables à sauvegarder :

```
->a=[1 2 3];
->function y=foo(a,b),y=a+b,endfunction
->save("mes_donnees",a,foo)
```

Remarquons, ici encore, que les fonctions sont des objets comme les autres.

La fonction `load` joue le rôle symétrique de la fonction `save`. Pour charger l'ensemble des variables stockées dans un fichier il suffit d'entrer l'instruction `load("fichier")`.
Si l'on ne veut charger que quelques variables à partir d'un fichier, il faut préciser le nom des variables désirées :

```
->load("mes_donnees") //recharge les variables a et foo
->load("mes_donnees","foo") //charge uniquement la variable foo
```

Noter qu'ici il faut donner le nom comme une chaîne alors que pour `save` il faut passer l'objet en argument.

4.5.2 Fonctions de sauvegarde des graphiques

Les fonctions `save` et `xsave`, `load` et `xload` permettent de sauvegarder et de recharger les graphiques de Scilab dans des fichiers binaires. Ces fonctions seront décrites au chapitre 5.

4.5.3 Fonctions de sauvegarde des fichiers de son

Les fonctions `savewave` et `loadwave` permettent la lecture et l'écriture des fichiers de son au format `wav`. Les fonctions `auwrite` et `auread` font de même pour les fichiers de son au format `au`.

4.5.4 Lecture de fichiers de données formatées

La lecture de fichiers de données formatées peut aussi se faire en utilisant la fonction `exec` dont on a parlé précédemment pour l'exécution des scripts Scilab. En effet il est souvent possible, en modifiant légèrement le fichier, de le transformer en un script Scilab. Par exemple, si le fichier se présente comme un tableau de nombres séparés par des blancs ou des virgules, en rajoutant une première ligne de la forme `A = [` et une dernière de la forme `];` on obtient un script qui définit une matrice `A` ayant autant de lignes que le fichier initial en avait.

4.6 Fonctions d'interaction homme-machine

Scilab permet de développer des applications interactives. Pour cela, il existe un certain nombre de fonctions pour l'interaction homme-machine, par dialogues, menus et souris.

4.6.1 Dialogues

Il existe dans Scilab des boîtes de dialogues programmables. La fonction de dialogue la plus simple est `x_message` qui permet d'afficher un texte dans une fenêtre (figure 4.2) :

```
->texte=["Coucou";"C""est moi, ton copain Scilab !"];

->x_message(texte)
```

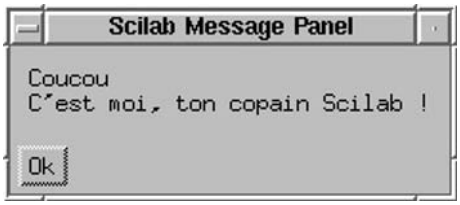


Figure 4.2 – La fonction `x_message`.

Le texte que l'on utilise dans `x_message` est un texte figé, non éditable par l'utilisateur. La fonction `x_message` est bloquante : l'interprète est suspendu jusqu'à ce que l'utilisateur ait cliqué sur le bouton `Ok`.

La fonction `x_message_modeless` affiche un message sans être bloquante.

La fonction `x_dialog` permet une interaction dans les deux sens par l'intermédiaire de la boîte de dialogue de la figure 4.3.

```
->texte=x_dialog(["Quel est votre age ?"],"18");
->age=evstr(texte)
```

En général, les fonctions de dialogue manipulent des chaînes de caractères ou des matrices de chaînes de caractères. Cela explique l'utilisation de la fonction `evstr` dans l'exemple précédent.

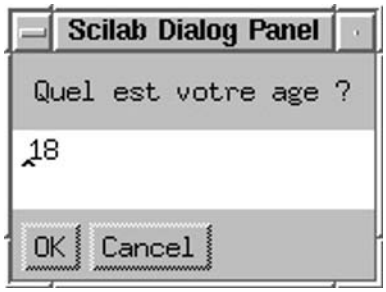


Figure 4.3 – La fonction `x_dialog`.

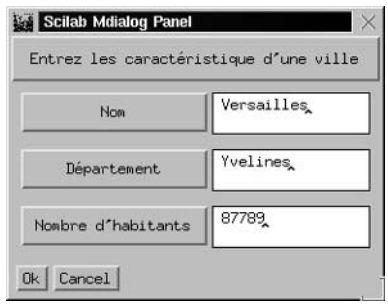


Figure 4.4 – La fonction `x_mdialog`.

Il existe d'autres fonctions de dialogue plus sophistiquées comme `x_mdialog` ou `getvalue` qui gèrent une fenêtre de dialogue comprenant plusieurs champs éditables comme celle de la figure 4.4 correspondant à l'instruction ci-dessous.

```
->x_mdialog("Entrez les caractéristiques d""une ville",..
            ["Nom";"Département";"Nombre d""habitants"],..
            ["";"";""])
 ans  =

!Versailles  !
!Yvelines    !
!87789       !
```

ou encore `x_choose` qui gère une fenêtre de sélection(figure 4.5) ou `x_choices` qui gère une fenêtre de sélections multiples :

```
->x_choose(["Scilab","Matlab","Octave","RLab"],..
               "Choisissez votre logiciel préféré")
```

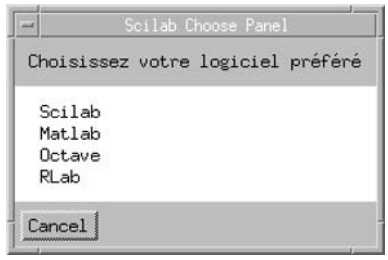


Figure 4.5 – La fonction `x_choose`.

4.6.2 Menus

La fenêtre principale et les fenêtres graphiques de Scilab sont pourvues de menus, déroulants ou non. Il est possible d'ajouter de nouveaux menus et des actions ou de supprimer les menus existants. Supposons que l'on veuille charger toutes les fonctions définies dans des fichiers du répertoire courant dont le nom se termine par `.sci` par un simple clic (ce que fait la fonction `getd`). Pour cela, il faut définir un bouton (menu non déroulant) et l'action associée à ce bouton.

```
->Charger="getd()"

->addmenu("Charger")
```

Lorsque l'on clique sur le bouton `Charger` créé par `addmenu`, Scilab exécute la commande `execstr(Charger(1))` pour réaliser l'action voulue.

Par défaut, la commande `addmenu` ajoute des menus dans la fenêtre principale. Pour placer des menus dans les fenêtres graphiques, il faut passer comme argument le numéro de la fenêtre graphique.

La fonction suivante adapte les menus de la fenêtre graphique 0 en supprimant des menus existants par défaut et en rajoutant un menu `Commandes`.

```
function mes_menus()
  if ~MSDOS then
    delmenu(0,"3D Rot.")
    delmenu(0,"UnZoom")
    delmenu(0,"2D Zoom")
  else
    delmenu(0,"3D &Rot.")
    delmenu(0,"&UnZoom")
    delmenu(0,"2D &Zoom")
  end
  addmenu(0,"Commandes",["Couleurs";"Precision";"quit"])
endfunction

Commandes_0=["flag=""Couleurs""";
             "flag=""Precision""";
             "flag=""quit"""]
```

Fichier source : `scilab/mes_menus.sci`

La variable booléenne `MSDOS` permet de tester si l'on se trouve sous le système d'exploitation Windows ou non. Les menus par défaut des fenêtres graphiques sont un peu différents sous le système Windows. Un `&` précédant un caractère indique que ce caractère est un raccourci-clavier correspondant à ce menu. Cette fonctionnalité n'est pas disponible sous Xwindow.

Lorsque le sous-menu `Couleurs` du menu déroulant `Commandes` est activé par un clic, la commande `execstr(Commandes_0(1))` est exécutée, ce qui revient à

affecter à la variable `flag` la chaîne de caractères `Couleurs`. De même pour les sous-menus `Precision` et `quit`. Si cela se produit pendant l'exécution d'une fonction, cette commande est exécutée dès que l'instruction courante est terminée. Nous verrons plus loin une utilisation de cette fonctionnalité.

La fonction `addmenu` peut être utilisée dans d'autres contextes ; en particulier, il est possible d'associer une fonction Scilab (ou même une fonction C ou FORTRAN) à chaque menu.

```
addmenu(0,"Commandes",["Couleurs";"Precision";"quit"],..
                        list(2,"mes_commandes"))

function mes_commandes(k,win)
  global flag
  select k
  case 1 then
    flag="Couleurs"
  case 2 then
    flag="Precision"
  case 3 then
    flag="quit"
  end
endfunction
```

Fichier source : `scilab/mes_menus2.sci`

Un menu peut être inhibé par la fonction `unsetmenu`, réactivé par la fonction `setmenu`, ou définitivement supprimé par la fonction `delmenu`.

4.6.3 Souris

Scilab peut récupérer certains événements souris provenant des fenêtres graphiques. Les deux principales fonctions utilisées à cet effet sont `xclick` et `xgetmouse`. Ces fonction seront décrites dans la section 5.6.

4.6.4 Interface TCL-TK

Scilab intègre une interface avec le langage TCL/TK qui permet de programmer des interfaces utilisateur très élaborées.[20][13][12]

Cinq fonctions forment l'essentiel de cette interface. La fonction `TCL_EvalStr` prend en argument un vecteur de chaînes de caractères contenant des instruction TCL et les évalue. L'instruction :

```
TCL_EvalStr(["toplevel .foo"
   "label .foo.l -text ""TCL married Scilab !!!"""
   "pack .foo.l"
   "button .foo.b -text close -command {destroy .foo}"
   "pack .foo.b"])
```

Figure 4.6 – Un exemple de fenêtre de message TCL/TK.

Figure 4.7 – Fenêtre de dialogue initiale.

Fichier source : `scilab/tcl_evalstr.sce`

produit la fenêtre présentée en figure 4.6.

La fonction `TCL_EvalFile` permet d'exécuter l'ensemble des instructions TCL/TK contenues dans un fichier dont on donne le chemin en argument.

La fonction `TCL_GetVar` prend en argument d'entrée le nom d'une variable TCL et retourne sa valeur dans l'environnement Scilab sous forme d'une chaîne de caractères ou d'une matrice de chaînes.

```
TCL_EvalStr(["toplevel .foo"
     "label .foo.l -text ""entrez un texte\ndans la zone de saisie"""
     "pack .foo.l"
     "entry .foo.e -textvariable tvar"
     "set tvar """""
     "pack .foo.e"])
```

Fichier source : `scilab/tcl_getvar.sce`

Cette séquence provoque l'ouverture de la fenêtre présentée dans la figure 4.7 et l'utilisateur saisit son texte (figure 4.8). Il est alors possible de récupérer le texte saisi dans Scilab, voire de l'évaluer s'il s'agit comme ici de syntaxe Scilab.

```
->text=TCL_GetVar("tvar")
 text  =

 A =[ 2 3 5]

->execstr(text)
```

Figure 4.8 – La saisie du texte.

```
->A
 A  =

!   2.    3.    5. !
```

La fonction `TCL_SetVar` réalise l'opération inverse et affecte à une variable TCL une valeur qui peut être donnée par une matrice de nombres ou de chaînes de caractères.

Enfin la fonction TCL `ScilabEval` permet d'exécuter une instruction Scilab dans un script TCL. L'instruction

```
TCL_EvalStr(["toplevel .file"
          "label .file.l -text ""Entrez  un nom de fichier"""
          "pack .file.l"
          "entry .file.path -textvariable tvar";
          "set tvar ""SCI/macros/elem/cosh.sci""";
          "pack .file.path";
          "button .file.edit -text ""edit"" "+..
                  "-command {ScilabEval ""scipad $tvar""}";
          "pack .file.edit"
          "button .file.getf -text ""getf"" "+..
                       "-command {ScilabEval ""getf $tvar"" }";
          "pack .file.getf"])
```

Fichier source : `scilab/scilabeval.sce`

produit la fenêtre présentée dans la figure 4.9

L'exemple ci-dessus illustre l'utilisation de la fonction `ScilabEval`. L'évaluation a lieu à l'instant et dans le contexte où l'utilisateur presse l'un des boutons. En particulier, si cette action a lieu alors que l'interpréteur Scilab est en train d'évaluer une instruction, l'évaluation est suspendue, la commande générée par `ScilabEval` est évaluée, puis l'interprétation de l'instruction en cours reprend.

Nous avons vu quelques exemples simples pour illustrer le fonctionnement de l'interface TCL/TK dans Scilab. Il est possible de réaliser des fenêtres de dialogue beaucoup plus élaborées en utilisant cette interface et la puissance du langage TCL/TK. À titre d'exemple l'éditeur de texte `scipad` intégré dans Scilab ainsi que l'éditeur de propriétés graphiques `ged` et la visualisation de

Figure 4.9 – Utilisation de `ScilabEval`.

l'aide en ligne sont réalisés en TCL/TK et utilisent les fonctions d'interfaces entre TCL et Scilab.

Des éditeurs graphiques permettent de développer graphiquement des fenêtres de dialogue homme-machine. On peut citer le logiciel libre et *open source* VTCL [3] qui fonctionne sous Unix/Linux et Windows.

[3]Consulter le site `vtcl.sourceforge.net`.

Chapitre 5

Graphiques

5.1 Introduction

La visualisation graphique est une fonctionnalité complexe du fait de la multiplicité des paramètres qu'il est possible d'ajuster pour obtenir le rendu souhaité. On donne ici quelques rudiments pour réaliser les tâches les plus courantes. Pour des graphiques plus sophistiqués, la lecture de l'aide en ligne est indispensable.

Les fonctions graphiques de Scilab sont basées sur la construction et la manipulation d'entités graphiques (objets). Il est ainsi possible de changer *a posteriori* l'aspect d'un graphique en modifiant les propriétés graphiques du dessin obtenu. Ces modifications peuvent être réalisée soit par des instructions Scilab, soit à l'aide de l'éditeur graphique intégré `ged`.

5.2 Premiers pas

Les fonctions graphiques fondamentales sont `plot2d` et `plot`. La fonction `plot` émule la fonction Matlab du même nom. L'ordre graphique le plus simple est `plot2d(x,y)` où x et y sont deux vecteurs de même dimension. Cela produit le graphe de la ligne brisée obtenue en reliant chaque point de coordonnée `(x(i),y(i))` au point suivant de coordonnée `(x(i+1),y(i+1))`. Les vecteurs `x` et `y` doivent donc avoir au moins deux composantes. Lorsque le vecteur `x` est omis, il est pris par défaut égal à `1:n` où `n` est la dimension du vecteur `y`. Ainsi, pour tracer le graphe de la fonction sinus sur $[0, 2\pi]$ avec 100 points, on utilisera la commande :

```
x=linspace(0,2*%pi,100);
plot2d(x,sin(x))
```

La fonction `plot2d` a pour effet d'ouvrir une fenêtre graphique (voir la figure 5.1) dans laquelle apparaît le graphe. L'aspect de la fenêtre (boutons,

Fenêtres graphiques

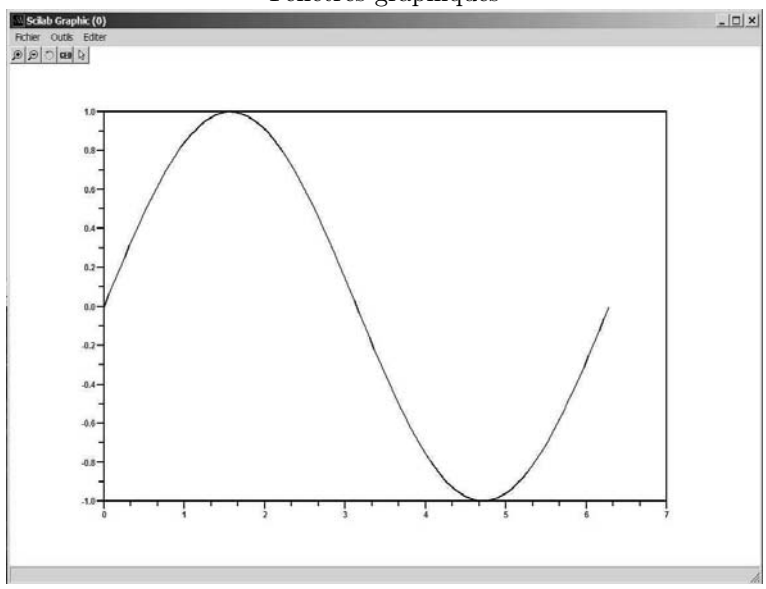


Figure 5.1 – Une courbe dans sa fenêtre graphique Windows.

scroll-bars...) peut être différent selon l'environnement (voir la figure 5.1 pour l'aspect sous Windows et la figure 5.2 pour l'aspect sous Unix).

On voit que la courbe tracée par la commande `plot2d` apparaît dans un cadre rectangulaire dont les côtés inférieur et gauche servent d'axes avec des graduations. La fenêtre graphique comporte aussi une barre de menus, et sous Windows, une barre d'outils[1].

Par défaut, la fonction `plot2d` n'efface pas la fenêtre graphique, mais elle superpose son tracé dans la fenêtre courante. Si, à la suite des commandes précédentes, on fait :

```
t=linspace(0,4*%pi,100);plot2d(t,0.5*cos(t))
```

on obtiendra (voir figure 5.3) le graphe des deux fonctions sinus et cosinus, dans une échelle commune (l'axe des x allant de 0 à 4π après le deuxième tracé).

Si l'on ne désire pas superposer les courbes, il est donc nécessaire de commencer par effacer la fenêtre graphique. Cela se fait par la commande `clf()` ou en cliquant sur le bouton `Effacer Figure` du menu `Éditer` de la fenêtre graphique (bouton `Clear` du menu `File` sous Unix).

Sur le graphe précédent, on peut aussi utiliser le menu `Zoom` pour zoomer sur une partie du graphe.

[1] On peut masquer la barre d'outils avec la fonction `hidetoolbar` et l'exposer avec la fonction `showtoolbar`.

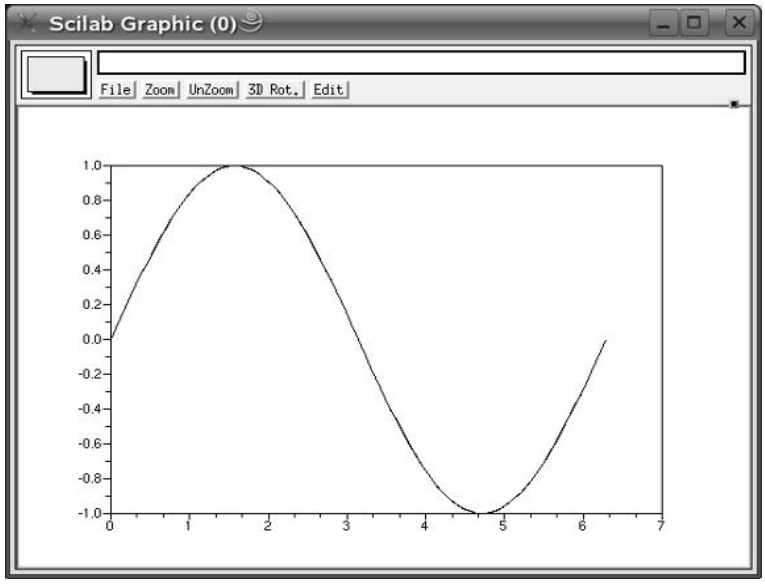


Figure 5.2 – Une courbe dans sa fenêtre graphique Unix.

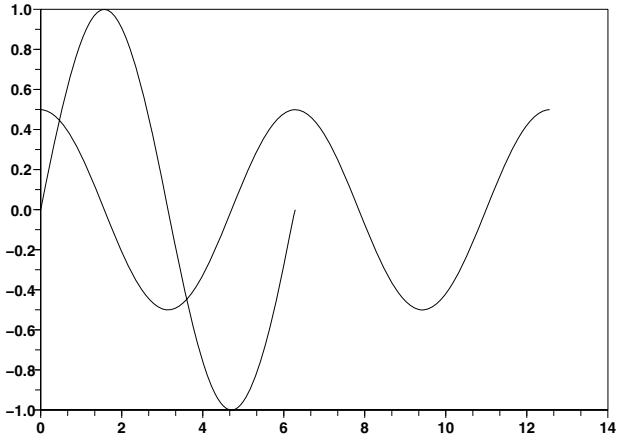


Figure 5.3 – Superposition de courbes.

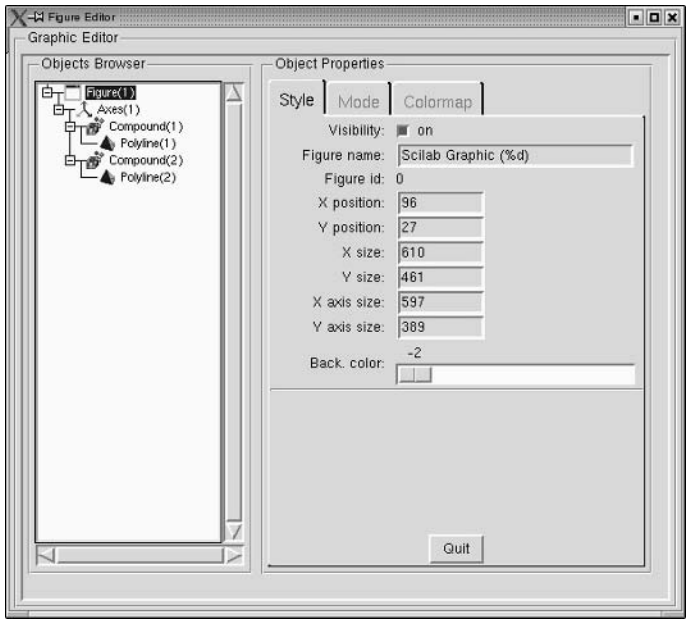


Figure 5.4 – Éditeur de propriétés graphiques.

5.3 Objets graphiques

5.3.1 Aperçu d'ensemble

Lorsque qu'une fenêtre graphique est créée, il lui est automatiquement associé un objet `Figure`. L'objet `Figure` est une structure de données hiérarchique qui contient l'ensemble des informations nécessaires à l'affichage des graphiques dans la fenêtre. Cette structure de données peut-être lue et modifiée depuis le langage Scilab ou grâce à l'éditeur de propriétés.

Pour modifier le graphique de la figure 5.3 on pourra utiliser l'éditeur graphique `ged`. Cet éditeur peut se lancer par le menu déroulant `Éditer` (`Edit` sous Unix). Ainsi par exemple, en cliquant sur l'item `Propriétés de la figure` on obtiendra la fenêtre de dialogue de la figure 5.4.

En navigant dans la hiérarchie des objets graphiques (voir la figure 5.7) grâce à l'`Object Browser` il est possible d'accéder aux propriétés des différents objets constituant le graphique. En sélectionnant l'objet `Polyline(1)` on obtient la fenêtre de dialogue de la figure 5.5.

En modifiant les champs `Foreground`, `Mark mode`, `Mark style`, `Mark size` la figure 5.3 se transforme en la figure 5.6.

Il est aussi possible d'accéder aux propriétés des objets graphiques par des instructions Scilab.

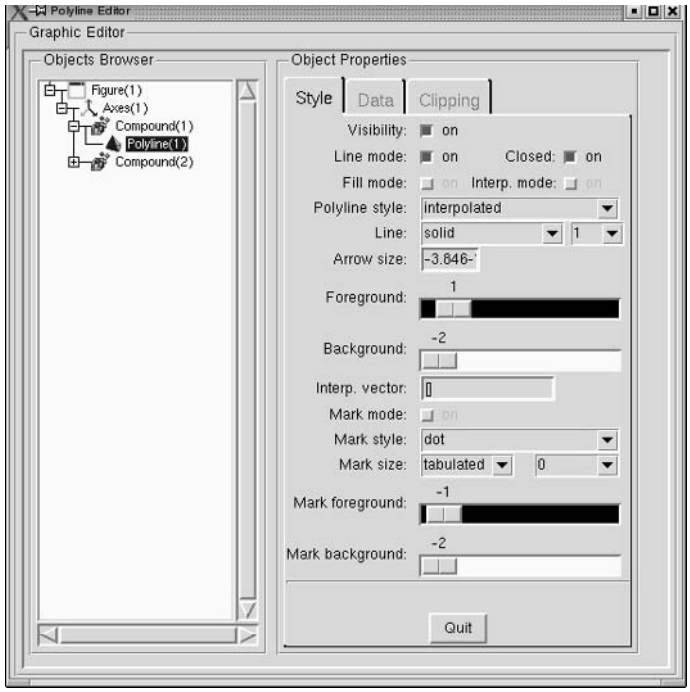


Figure 5.5 – Éditeur de propriétés des `polylines`.

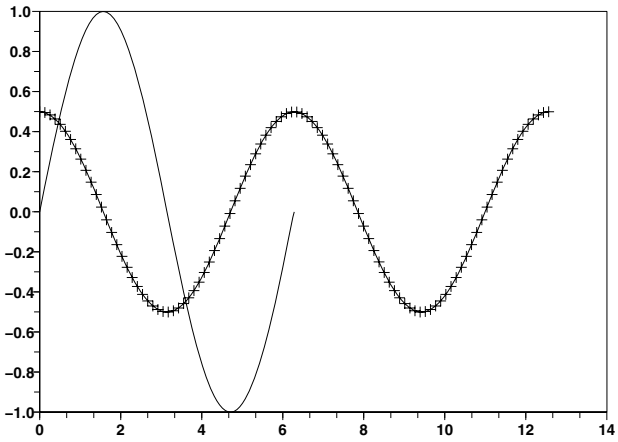


Figure 5.6 – Graphique modifié par l'éditeur.

Dans l'environnement de programmation les objets sont représentés par leurs `handles`[2]. Le handle associé à la fenêtre graphique courante peut être obtenu par l'instruction `f=gcf()` ou, de façon équivalente, par l'instruction `f=get("current_figure")`. La variable `f` est une variable de type structure (handle) dont les champs (propriétés) décrivent les caractéristiques de la fenêtre courante. On aura ainsi par exemple pour la fenêtre de la figure 5.3 :

```
-> f=gcf()
 f  =
Handle of type "Figure" with properties:
========================================
children: "Axes"
figure_style = "new"
figure_position = [981,303]
figure_size = [610,461]
axes_size = [600,400]
auto_resize = "on"
figure_name = "Scilab Graphic (\%d)"
figure_id = 0
color_map= matrix 32x3
pixmap = "off"
pixel_drawing_mode = "copy"
immediate_drawing = "on"
background =  -2
visible = "on"
rotation_style = "unary"
```

Il est possible d'accéder aux champs avec les fonctions `set` et `get`. Par exemple, il est possible d'obtenir la taille de la figure, exprimée en pixels par l'instruction : `get(f, "figure_size")` qui retournera un vecteur à deux composantes, soit sur cet exemple `[610,461]`. Cette instruction peut aussi être écrite sous la forme plus condensée : `f.figure_size` que l'on utilisera de préférence dans la suite.

L'instruction `f.figure_size(1)=1000` permet d'élargir la figure à 1000 pixels. On peut noter ici que les couleurs utilisables sont représentées par une table de couleurs `f.color_map` qui est par défaut une matrice à 32 lignes (nombre de couleurs) et 3 colonnes (codage Rouge Vert Bleu RVB de la couleur). Par convention, le numéro de couleur `-2` représente le blanc et `-1` représente le noir. On aura plus de détails et des exemples d'utilisation des différents champs de `f` par les commandes `help graphics_entities` et `help figure_properties`.

On remarquera le champ `children` de la structure `f`. Ce champ est lui même une structure de type `Axes` qui contient en particulier la définition du changement de coordonnées entre les données utilisateur et les pixels de l'écran. On peut récupérer le handle sur cet objet soit la commande `a=f.children`, soit par

[2] Alors que les objets standards de Scilab sont passés par valeur dans les appels de fonction, les variables de type `handle` permettent d'accéder à ces variables par référence. Une fonction peut ainsi modifier la valeur d'un objet graphique sans qu'il apparaisse dans le membre de gauche.

`a=gca()` (get current axes). On voit aussi sur cet exemple que `a.children` est un vecteur de deux objets de type `Compound` et que enfin `a.children(1).children` est une structure de type `Polyline` dont le champ `data` contient les données tracées. On pourra ainsi modifier l'aspect de la courbe, ou même faire une animation en changeant les valeurs du champ `data`.

On voit sur cet exemple et sur la figure 5.7 comment on peut accéder de manière hiérarchique aux différentes entités graphiques.

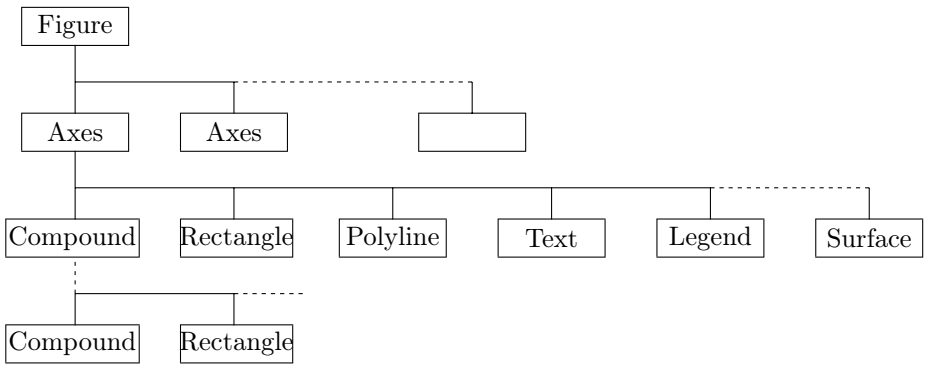


Figure 5.7 – `Figure`, `Axes`, ..., objets graphiques.

La commande `plot2d` ajoute une sous-hiérarchie `Compound->Polyline` aux enfants de l'objet `Axes` courant car cette fonction de haut niveau peut produire plusieurs objets de type `Polyline`. Au contraire, la fonction `xpoly`, qui dessine une ligne brisée, créera directement un objet de type `Polyline` enfant de l'objet `Axes` courant.

Les propriétés graphiques sont héritées dans le passage `parent->children`, mais on notera que les axes et la figure ne partagent pas de propriétés communes.

La commande `delete(h)`, où `h` représente un handle graphique, détruit l'objet graphique désigné par `h` ainsi que tous ses descendants.

Une autre fonction utile qui agit directement sur un handle est la fonction `move`. Donnons un exemple simple. On trace un petit rectangle dans une fenêtre et on le translate. Pour réaliser cette opération, il faut récupérer le handle graphique sur l'objet graphique que l'on désire translater, puis lui appliquer la commande `move`.

```
a=gca();  //on cree des axes et eventuellement une figure
//par défaut les bornes des coordonnées utilisateur sont
// [0 1] pour x et y
x=[2;3;3;2;2]/10;
y=[2;2;3;3;2]/10;
plot2d(x,y);   // le rectangle.
```

```
f=gcf();  // handle de la figure.
mypolyline=f.children.children.children;  //handle du rectangle
move(mypolyline,[0.2,0.2]);
```

Fichier source : `scilab/move.sce`

Sur cet exemple on voit que pour accéder à l'objet de type `Polyline` créé par la commande `plot2d` il faut parcourir le chemin :

```
Figure->Axes->Compound->Polyline
```

comme cela apparaît lorsqu'on utilise sur l'éditeur graphique. Ce type de chemin est celui qu'on obtient naturellement dès qu'on exécute la commande `plot2d`. On aurait pu accéder directement et plus simplement au handle de l'objet `Polyline` à partir de l'instruction :

```
e=gce();mypolyline=e.children;
```

l'instruction `gce()` (ou `get("current_entity")`) retournant le handle sur l'objet courant, ici l'objet `Compound`.

On aurait pu aussi utiliser plus simplement l'instruction :

```
move(gce(),[2,2])
```

pour translater l'objet composite `Compound` et l'ensemble de ses enfants.

On notera que la commande `move` ne fait que modifier les valeurs du champ `mypolyline.data`. La translation du rectangle aurait donc pu être faite aussi par la commande :

```
mypolyline.data=mypolyline.data+2;
```

(`mypolyline.data` n'est autre que la matrice 5x2 `[x,y]`). Cet exemple montre comment on pourrait programmer en Scilab la fonction `move` en modifiant le champ `data` de l'objet `Polyline`.

5.3.2 Figure et fenêtre graphique

Une fenêtre graphique est créée automatiquement lors de l'appel d'une fonction graphique, si aucune fenêtre graphique n'existe. La fonction `scf` (set current figure) permet de créer explicitement une fenêtre graphique et de la désigner comme fenêtre courante. La commande `clf()` (clear figure) conserve les paramètres courants de la figure mais détruit tous les objets `Axes` enfants.

La `Figure` est le premier objet graphique de la hiérarchie d'objets associés à une fenêtre. Lors de sa création l'objet `Figure` hérite d'un objet particulier (`Default Figure`) qui comprend les mêmes champs mais n'est jamais affiché. Le handle de cet objet modèle peut être obtenu par la fonction `gdf` (get default figure), ce qui permet dans modifier les propriétés. Nous présentons ci-dessous les propriétés essentielles de ces objets de type `Figure`.

Table des couleurs

Chaque figure peut être associée une table de couleurs différente. Une table de couleurs à n couleurs est donnée par une matrice à n lignes et trois colonnes représentant les trois couleurs fondamentales rouge, vert, bleu. Chaque ligne définit une couleur à partir des couleurs fondamentales en donnant pour chacune un nombre compris entre 0 et 1. Ainsi, la table définie par `T=eye(3,3)` produit une table à trois couleurs, rouge (numéro 1), vert (numéro 2) et bleu (numéro 3). Scilab ajoute à la table des couleurs le noir `[0,0,0]` et le blanc `[1,1,1]`.

Les fonctions `graycolormap`, `hotcolormap`, `hsvcolormap` et `jetcolormap` définissent des tables de couleurs particulières.

Les instructions :

```
f=gcf();
f.color_map=hotcolormap(128)
```

affectent à la fenêtre `f` une table de couleurs comprenant 128 couleurs allant du jaune très clair au rouge très foncé.

Couleur de fond

La propriété `background` de la fenêtre fixe l'index de la couleur de remplissage du fond dans la table des couleurs. La valeur par défaut est `-2` qui représente le blanc.

Fenêtre virtuelle, fenêtre visible

Les ordres graphiques dessinent dans une zone mémoire appelée fenêtre virtuelle qui est visualisée à travers la fenêtre graphique visible (une partie de l'écran). Par défaut, la fenêtre virtuelle remplit complètement la fenêtre visible ; même lorsque l'on redimensionne cette dernière, la fenêtre virtuelle continue à remplir toute la fenêtre visible.

L'instruction `f=gcf();f.auto_resize='off';` permet de dissocier ces deux fenêtres. Le mode par défaut est `f.auto_resize='on'`.

En haut et à gauche de la fenêtre graphique, dans l'environnement Xwindow, on trouve un «panner» constitué de deux rectangles. Le rectangle intérieur représente la fenêtre visible et le rectangle extérieur la fenêtre virtuelle dans laquelle on dessine. Sous Windows ce panner est remplacé par deux barres de défilement (horizontale et verticale). Dans le mode par défaut `f.auto_resize='on'` ces deux rectangles coïncident et les barres de défilement sont inactives.

On peut régler la largeur `L` et la hauteur `H` (en pixels) de la fenêtre visible par l'instruction `f.figure_size = [L,H]`. L'instruction `f.axes_size=[l,h]` permet de spécifier les dimensions de la fenêtre virtuelle. En mode `f.auto_resize='on'`, la spécification de l'un ou l'autre de ces paramètres `figure_size` ou `axes_size` modifie l'autre. En mode `f.auto_resize='off'`, les deux rectangles ne sont plus solidaires et si la taille de la fenêtre virtuelle est plus grande que la taille de la

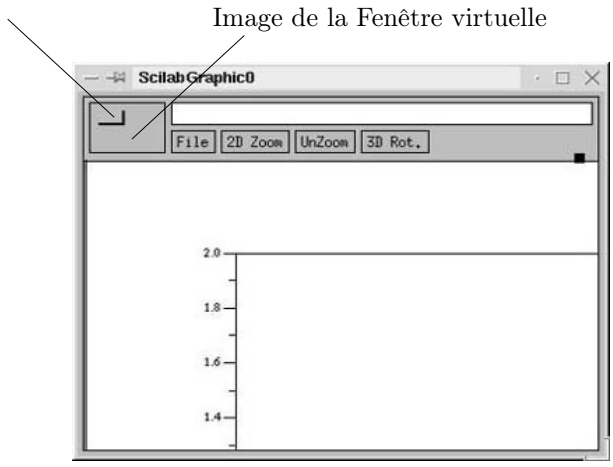


Figure 5.8 – Panner : on sélectionne la partie visualisée (Xwindow).

fenêtre visible, on ne voit dans la fenêtre visible qu'une partie du dessin tracé. On peut voir n'importe quelle partie du dessin tracé dans la fenêtre virtuelle en faisant glisser, avec le panner ou les barres défilements, l'image de la fenêtre visible. Dans l'exemple suivant, on réalise un tracé avec `plot2d`, dans une fenêtre virtuelle de 900×600 pixels, placée dans une fenêtre visible de taille 300×200 pixels. On ne voit dans la fenêtre graphique qu'une partie du dessin tracé mais on peut explorer l'ensemble du dessin en faisant glisser l'image de la fenêtre visible dans le panner : voir la figure 5.8.

```
f.auto_resize='off'     //On désolidarise les fenêtres
//dimension de la fenêtre visible (pixels)
f.figure_size=[300,200];
//dimension du la fenêtre visible virtuelle
f.axes_size=[900,600];
plot2d(0:2,0:2) //On dessine la droite y=x, x dans [0,2]
```

Fichier source : `scilab/autoresize.sce`

Sélection d'une fenêtre/Figure

Il est possible de gérer plusieurs fenêtres graphiques. les ordres graphiques sont envoyés à la fenêtre graphique courante que l'on peut positionner avec la fonction `scf` : `scf(n)` positionne la fenêtre numéro `n` comme courante et `scf(f)` positionne la fenêtre dont le handle est `f` comme courante.

La fonction `winsid` retourne le vecteur des numéros des fenêtres graphiques ouvertes.

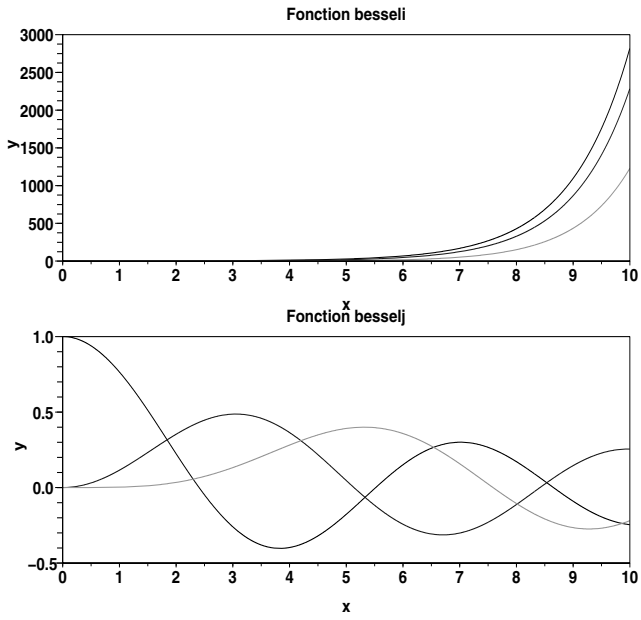


Figure 5.9 – fonctions de Bessel et gestion des `Axes`.

5.3.3 Axes

Chaque `Figure` comporte au moins un objet `Axes` qui est automatiquement créé a la création de la figure ou lorsque le dernier objet `Axes` est détruit.

Propriétés par défaut

Lors de sa création, l'objet `Axes` hérite d'un objet particulier (`Default Axes`) qui comprend les mêmes champs mais n'est jamais affiché ; le handle de cet objet modèle peut être obtenu par la fonction `gda` (get default axes).

Sélection d'un système d'axes

Une figure peut posséder plusieurs axes. Les fonctions `subplot` et `newaxes` permettent de créer de nouveaux systèmes d'axes. Pour sélectionner le système d'axes courant on utilisera l'instruction `sca(h)`, où `h` est un handle sur un objet `Axes`. Voici un exemple d'utilisation de la fonction `sca` qui produit la figure 5.9 :

```
x = linspace(0.01,10,5000)';
subplot(2,1,1);a1=gca(); //moitié supérieure de la fenêtre
plot2d(x,besseli([0 2 4],x))

subplot(2,1,2);  a2=gca(); //moitié inférieure de la fenêtre
```

```
plot2d(x,besselj([0 2 4],x))
sca(a1); // On remet les premiers axes comme axes courant.
//ajout du titre et des labels
title('Fonction besseli')
xlabel('x');ylabel('y')

sca(a2); // On remet le second  axes comme axes courant.
title('Fonction besselj')
xlabel('x');ylabel('y')
```

Fichier source : `scilab/subplot.sce`

Ici on trace deux courbes et on sélectionne comme `Axes` courant le premier système de coordonnées pour ajouter titre et légendes d'axes au premier graphique.

L'instruction `subplot(m,n,k)` (qui peut être abrégée en `subplot(mnk)` si `m`, `n` et `k` sont des chiffres) découpe virtuellement la fenêtre graphique en une matrice de sous-fenêtres à `m` lignes et `n` colonnes et crée le système de coordonnées `Axes` associé à la sous-fenêtre `k=i+(j-1)*m`.

Propriétés de l'objet Axes

L'objet `Axes` contient de très nombreuses propriétés que l'on peut classer en 3 grandes catégories :

- Certaines (voir la table 5.1) définissent le changement de coordonnées à appliquer aux coordonnées utilisateur pour les transformer en coordonnées pixel de la fenêtre virtuelle. `axes_bounds` (`[xleft,yup,width,height]`) définit le positionnement du rectangle associé au système d'axes à l'intérieur de la fenêtre graphique. Les grandeurs `xleft`, `yup`, `width`, `height` sont données en pourcentage des dimensions de la fenêtre graphique. Le dessin des graduations est positionné à l'intérieur de ce rectangle par l'intermédiaire de la propriété `margins` qui définit les 4 marges (gauche, droite, haute et basse) données en pourcentage des dimensions du rectangle.

- D'autres (voir la table 5.2) définissent la manière dont les graduations, les titres et les légendes sont dessinés. Les propriétés `x_ticks`, `y_ticks`, `z_ticks` sont des structures de type `tlist` comprenant deux champs : `locations` et `labels`. Le champ `locations` est un vecteur de nombres définissant la position des graduations le long de l'axe, tandis que le champ `labels` contient les chaînes de caractères associées à ces graduations. Dans le cas (défaut) où la propriété `auto_ticks` est égale à `"on"` les positions des graduations de l'axe correspondant ainsi que les légendes sont déterminés automatiquement.

Propriété	Définition
`isoview`	Mode isométrique
`cube_scaling`	Projection 3D dans un cube
`view`	Sélection 2D/3D
`rotation_angles`	Angles de vue en 3D
`log_flags`	Affichage `n`ormal ou `l`ogarithmique
`tight_limits`	Choix de limites strictes ou non
`data_bounds`	Bornes des coordonnées utilisateur
`axes_reverse`	Inversion des axes
`axes_bounds`	Rectangle support en pourcent de la fenêtre
`margins`	Marges en pixel
`zoom_box`	Rectangle zoomé
`auto_clear`	Effacement automatique
`auto_scale`	Adaptation automatique des bornes
`clip_state`	Mode de clipping
`clip_box`	Rectangle de clipping

Table 5.1 – Propriétés associées au changement de coordonnées.

– D'autres enfin sont des valeurs de paramètres dont pourront hériter les objets enfants.

Le script suivant et la figure 5.10 présentent des exemples de paramétrage de l'objet `Axes` :

```
//propriétés de la fenêtre
clf();f=gcf();f.background=color('gray');
//proprietés des axes par défaut
da=gda(); //handle de l'axe par défaut
da.font_style = 8;//helvetica bold
da.grid = [-1,color('green')];//Quadrillage en y
da.font_size = 2;//6 tailles disponibles indexées de 1 à 6

subplot(221) //En haut a gauche
plot2d2(rand(1,24,'normal')) //tracé sous forme échelon
a1=gca();
a1.thickness=2; //épaisseur des traits d'axes
a1.box = "off"; //pas de boîte englobant les axes.

a1.font_style = 8; //helvetica bold
x_ticks = a1.x_ticks;//une tlist
x_ticks.locations = 0:4:24; //une heure sur 4
x_ticks.labels = string(x_ticks.locations)+"h";
a1.x_ticks = x_ticks;
a1.y_location = "right";
a1.x_label.text = "Heures";a1.y_label.text = "Valeurs";
```

Propriété	Définition
`visible`	Les enfants sont-ils dessinés
`axes_visible`	Les graduations sont-elles dessinées
`grid`	Affichage et couleur du quadrillage
`x_location`	Position de l'axe des x
`y_location`	Position de l'axe des y
`title`	Propriétés du titre
`x_label`	Propriétés du label de l'axe des x
`y_label`	Propriétés du label de l'axe des y
`z_label`	Propriétés du label de l'axe des z
`auto_ticks`	Positionnement automatique des graduations
`sub_ticks`	Nombre de sous-intervalles
`x_ticks.locations`	Positions des graduations sur l'axe des x
`y_ticks.locations`	Positions des graduations sur l'axe des y
`z_ticks.locations`	Positions des graduations sur l'axe des y
`x_ticks.labels`	Labels des graduations de l'axe des x
`y_ticks.labels`	Labels des graduations de l'axe des y
`z_ticks.labels`	Labels des graduations de l'axe des z
`box`	Existence d'une boîte encadrant
`foreground`	Couleur d'affichage des axes et graduations
`background`	Couleur de fond des axes
`thickness`	Épaisseur des traits

Table 5.2 – Propriétés associées à l'aspect.

```
a1.title.text = "Exemple";a1.title.font_size = 2;

subplot(223) //En bas a gauche
t = linspace(0,2*%pi,50);
a2 = gca();
a2.isoview = "on"; //mode isométrique
plot2d(sin(t),cos(t))
a2.sub_ticks = [0,0];//supression des sous-graduations

a3 = newaxes();
//axes_bounds en pourcentage de la fenêtre
xleft = 1/2; yup = 0;width = 1/2;height = 1 //à droite
a3.axes_bounds = [xleft,yup,width,height];
t = linspace(-10,10,100);
plot2d(t,(0.25+sinc(t))^2)
a3.log_flags = 'nl';// Axes logarithmique pour y
```

Fichier source : `scilab/axes_exemple.sce`

Scilab gère 9 fontes différentes, numérotées de 1 à 9, données dans la table 5.3. Chacune de ces fontes peut prendre 6 tailles différentes numérotées de 1 à 6.

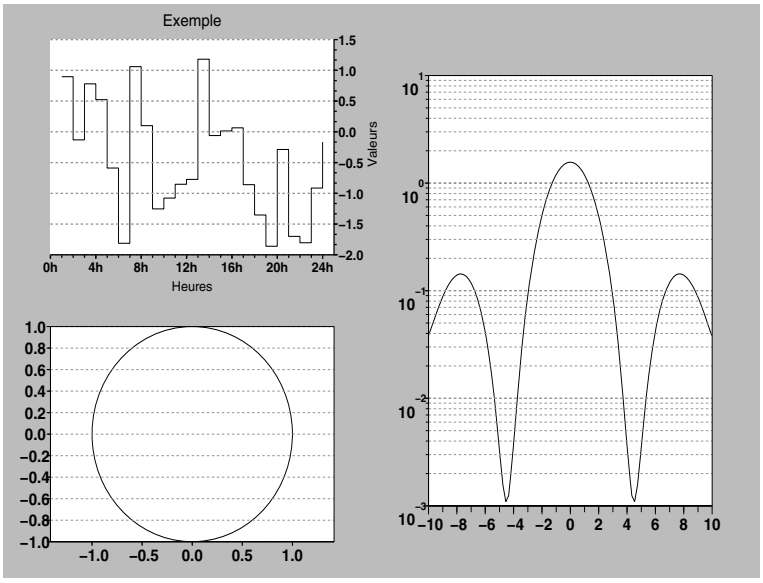


Figure 5.10 – Paramétrage des `Axes`.

Nom de fonte	N	Nom de fonte	N	Nom de fonte	N
`Courier`	1	`Symbol`	2	`Times`	3
`TimesItalic`	4	`TimesBold`	5	`TimesBoldItalic`	6
`HelveticaItalic`	7	`HelveticaBold`	8	`HelveticaBoldItalic`	9

Table 5.3 – Les fontes et leurs numéros.

La fonction `getfont` permet de choisir visuellement une fonte et sa taille.

Enfants

Les enfants de l'objet `Axes` sont les objets graphiques de base qui peuvent être de type `Polyline`, `Text`, `Segments`,...

5.3.4 Polylines

L'objet `Polyline` est l'objet de base du tracé de courbes ; avec ses différentes options il peut être utilisé pour représenter les données 2D et 3D sous les formes les plus diverses. Le script suivant, qui produit la figure 5.11, en donne les caractéristiques essentielles.

```
t=linspace(0,%pi,15);

subplot(311)
xmin=0;ymin=-0.5;xmax=4;ymax=2;
a=gca();a.data_bounds=[xmin,ymin;xmax,ymax];a.axes_visible='on';

//interpolation linéaire
xpoly(t,1+sin(t));xstring(3.3,1,'polyline_style=1')

//interpolation constante
xpoly(t,0.7+sin(t));
e=gce();e.polyline_style=2;
xstring(3.3,0.7,'polyline_style=2')

//interpolation linéaire et vecteurs
xpoly(t,0.4+sin(t));
e=gce();e.polyline_style=4;e.arrow_size_factor=3;
xstring(3.3,0.3,"polyline_style=4")

//barres
xpoly(t,sin(t));
e=gce();e.polyline_style=3;
xstring(3.3,0,"polyline_style=3")

subplot(312)
a=gca();a.data_bounds=[0 -1.5;4 1.5];a.axes_visible="on";

//polygone peint
xpoly(t,0.3+sin(t));
e=gce();e.polyline_style=2;e.thickness=2;
e.fill_mode="on";e.background=color("lightgray");
xstring(3.3,0.3,"fill_mode=""on""")

//barres
xpoly(t,-sin(t))
e=gce();e.polyline_style=6;e.line_mode="off";e.bar_width=0.1;
xstring(3.3,-0.1,"polyline_style=6")

subplot(313) //3D
xmin=0;ymin=-1.5;zmin=-1;xmax=4;ymax=1.5;zmax=1;
a=gca();a.data_bounds=[xmin, ymin,zmin;xmax,ymax,zmax];
a.axes_visible="on";
xpoly(t,sin(t))
e=gce();e.data(:,3)=cos(t')
```

Fichier source : `scilab/polyline.sce`

On remarquera l'utilisation de la fonction `xpoly`, qui crée un objet `Polyline`, et de la fonction `xstring`, qui permet d'afficher un texte en un point spécifié.

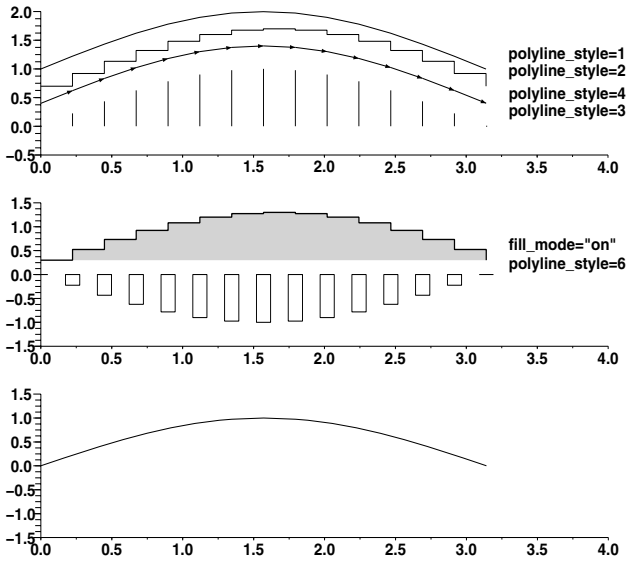


Figure 5.11 – Différentes facettes de l'objet `Polyline`.

5.3.5 Les objets graphiques

On a vu que les objets graphiques les plus communs de Scilab sont `Figure`, `Axes`, `Compound` et `Polyline`. La table 5.4 donne la liste de tous les autres objets graphiques existant dans Scilab ainsi que les fonctions permettant de les créer. L'aide en ligne de ces fonctions permet d'en connaître la syntaxe précise et fournit des exemples d'utilisation.

5.4 Principales fonctions graphiques

5.4.1 Visualisation des courbes

Les fonctions de visualisation des courbes sont construites sur la base de l'objet `Polyline` et en exploitent les différentes propriétés. Ces fonctions de haut niveau comme `plot`, `plot2d` gèrent aussi les propriétés de changement de coordonnées pour adapter automatiquement les bornes des coordonnées utilisateurs aux données si la propriété `auto_scale` du système d'axes associé est positionnée à `"on"` et lors de la superposition des ordres graphiques si la propriété `auto_scale` du système d'axes associé est positionnée à `"off"`.

Arc	xarc, xarcs
Axes	newaxes, subplot
Axis	drawaxis
Champ	champ
Compound	glue, plot2d, plot
Fac3d	surf,plot3d
Figure	scf
Grayplot	grayplot
Fec	fec
Legend	plot2d avec l'argument optionnel leg
Matplot	Matplot
Plot3d	plot3d
Polyline	xpoly, plot2d, plot, bar, param3d
Rectangle	xrect,xrects,
Text	xstring
Title	automatique
Label	automatique
Segs	xsegs

Table 5.4 – Les principales entités graphiques.

plot2d

La fonction `plot2d`, dont on a vu précédemment des exemples d'utilisation simples, permet de tracer une ou plusieurs courbes planes données par une discrétisation des abscisses et des ordonnées. La syntaxe d'appel de la fonction `plot2d` est :

`plot2d(x,y [,style])` ou `plot2d(x,y [,style=s])`

L'aide en ligne signale d'autres arguments optionnels qui peuvent être ignorés car aisément remplaçables par l'édition des paramètres des axes.

Les arguments `x` et `y` peuvent être des vecteurs ou des matrices (dans le cas de matrices, chaque colonne correspond à une courbe). Le paramètre optionnel `style` est un vecteur de même taille que le nombre de courbes, il permet d'indiquer la couleur ou le type de symbole utilisé pour dessiner la courbe correspondante

Lorsque le paramètre `style` est un entier positif, il représente la couleur choisie ; s'il est négatif il désigne un symbole. Ainsi le script suivant produit la figure 5.12 :

```
clf()
x=linspace(0,5,40)';//vecteur colonne
a=gca();
//propriétés préfixées
a.mark_background=color('lightgrey');// couleur marques
```

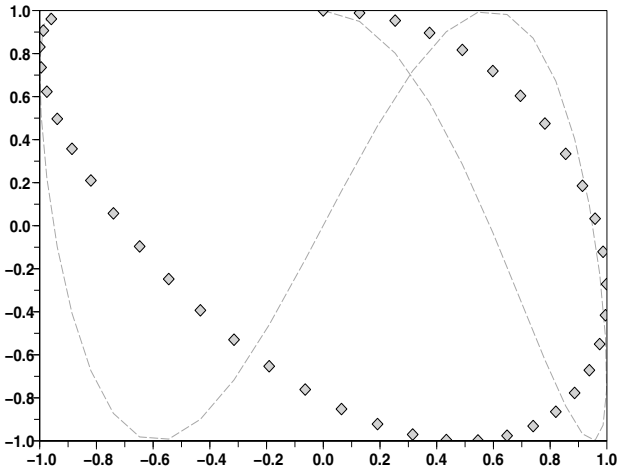


Figure 5.12 – Exemple d'utilisation de `plot2d`.

```
a.line_style=2;// pointillé
//deux courbes ayant les même coordonnées en x
cl=color('darkgrey'); //couleur gris foncé
plot2d(sin(x),[cos(2.5*x), cos(1.2*x)],style=[cl,-5])
```

Fichier source : `scilab/plot2d.sce`

L'instruction `color("nom_de_couleur")` renvoie le numéro de couleur associé à une couleur donnée par son nom anglais après l'avoir si nécessaire ajouté à la table des couleurs. Ainsi `color("brown")` retourne le numéro de la couleur marron dans la table. La liste des noms de couleurs disponibles est donnée par l'aide sur le mot `color_list`.

Les fonctions `getlinestyle` et `getmark` permettent de visualiser les types de traits et de symboles disponibles ainsi que les numéros associés...

Les paramètres positionnables par la fonction `plot2d` sont limités. Pour les autres, il faut modifier les propriétés des objets graphiques (`Polylines`, `Axes`) comme cela a été présenté précédemment.

La fonction `plot2d` admet une variante `fplot2d` qui permet de tracer une courbe d'équation $y = f(x)$ donnée sous forme analytique par une fonction Scilab. L'instruction suivante produit la figure 5.13 :

```
fplot2d(linspace(-10,10,100),sinc)
```

D'autres variantes de `plot2d` permettent de tracer des données sous forme échelon (`plot2d2`) ou sous forme de barres verticales (`plot2d3`). Ces variantes cor-

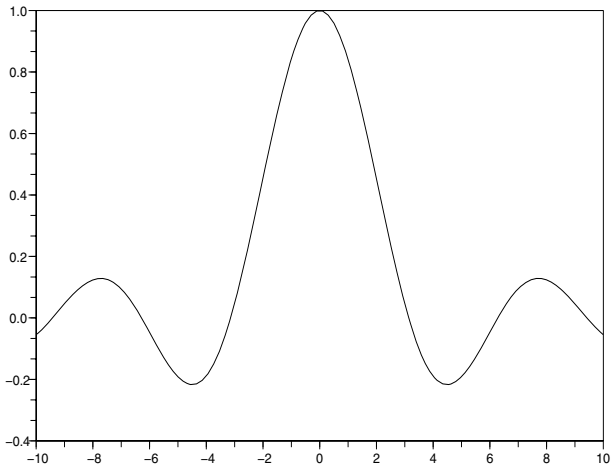


Figure 5.13 – Exemple d'utilisation de `fplot2d`.

respondent uniquement à un changement de la propriété `polyline_style` des objets `Polyline` générés.

plot

La fonction `plot` est une autre fonction pour l'affichage des courbes 2D compatible avec la fonction du même nom du logiciel Matlab. Le script suivant produit une figure similaire à la figure 5.12 :

```
x=linspace(0,5,40)';//vecteur colonne
lightgrey=[1 1 1]*211/255;
darkgray=[1 1 1]*169/255;
plot(sin(x),cos(2.5*x),'-k',sin(x),cos(1.2*x),'diamondk',..
                                  'MarkBackground',lightgrey)
```

Fichier source : `scilab/plot.sce`

bar

La fonction `bar` construite sur l'objet `Polyline` (avec `polyline_style = 6`) permet de visualiser des séries de données sous formes de barres verticales (voir `barh` pour les barres horizontales). Le script suivant génère la figure 5.14 :

```
clf()
// 3 series de données
```

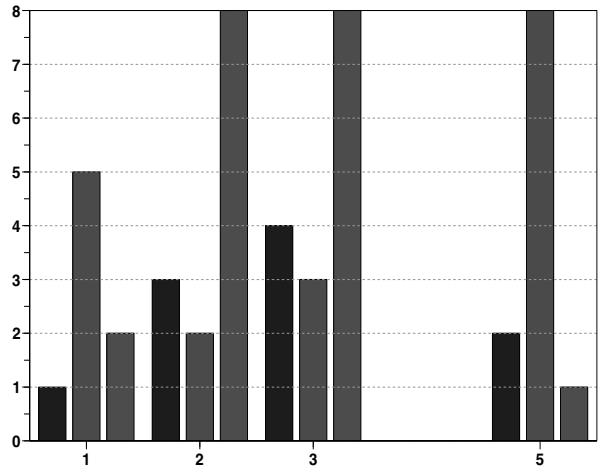


Figure 5.14 – Exemple d'utilisation de `bar`.

```
x=[1 2 3 5];  //abscisses
y=[1 3 4 2;  //série 1
   5 2 3 8;  //série 2
   2 8 8 1]';//série 3
bar(x,y);
a=gca();a.grid=[-1,3];
```

Fichier source : `scilab/bar.sce`

polarplot

La fonction `polarplot` permet de tracer des courbes en coordonnées polaires comme sur la figure 5.15. Cette figure est obtenue par :

```
t=0:0.01:2*%pi;
polarplot([sin(7*t') sin(6*t')], [cos(8*t') cos(8*t')]);
```

Fichier source : `scilab/polarplot.sce`

param3d

La fonction `param3d` permet de dessiner des courbes dans un univers tridimensionnel comme le montre l'exemple de la section 5.3.4.

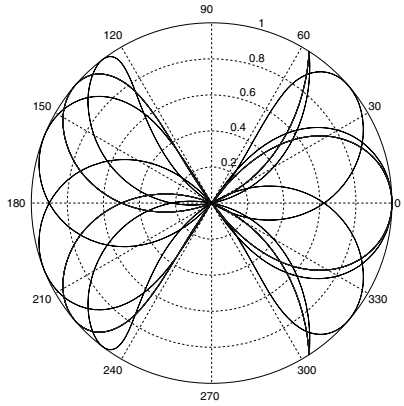


Figure 5.15 – `polarplot` : coordonnées polaires.

5.4.2 Visualisation 3D des surfaces

Les fonctions les plus utilisées pour visualiser les surfaces sont `plot3d` et `surf`.

la fonction plot3d et ses variantes

La fonction `plot3d` permet de représenter une fonction réelle de deux variables ou une surface. L'appel le plus simple de `plot3d` est `plot3d(x,y,Z)` où `x` est un vecteur (colonne) à `nx` composantes, `y` un vecteur à `ny` composantes et `Z` une matrice à `nx` lignes et `ny` colonnes. Cette commande donne une vue en perspective de la surface passant par les `nx*ny` points de coordonnées `(x(i), y(j), Z(i,j))`. Par exemple, considérons (voir la figure 5.16) la fonction en cloche $z(x, y) = \exp(-x^2 - y^2)$ où (x, y) désigne un point du plan et traçons cette fonction pour des valeurs de (x, y) dans le carré $[-2, 2] \times [-2, 2]$:

```
//surface définie analytiquement
function z=f(x,y),z=exp(-x^2 -y^2);endfunction
//calcul d'une discrétisation sur une grille
x=linspace(-2,2,50);y=x;
Z=feval(x,y,f);

plot3d(x,y,Z);   //On trace la cloche
e1=gce();e1.color_mode=color("gray95");
```

Fichier source : `scilab/plot3d_1.sce`

On peut superposer à la surface le plan tangent en `x0,y0`.

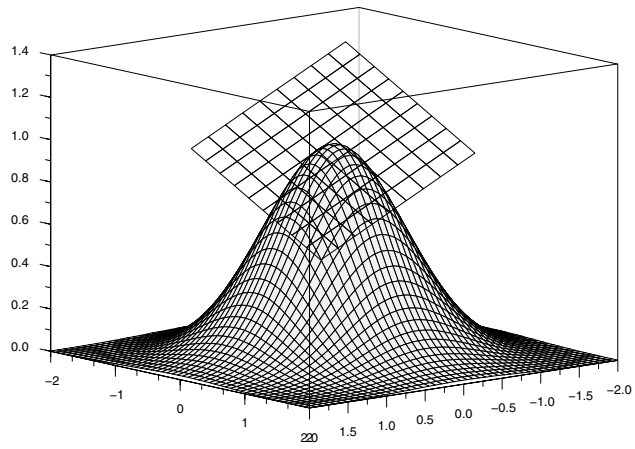


Figure 5.16 – `plot3d` : tracé d'une surface.

```
//équation du plan tangeant à f au point (x0,y0):
function dz=fp(dx,dy)
   dz=f(x0,y0)+exp(-x0^2-y0^2)*[-2*x0,-2*y0]*[dx;dy];
endfunction

x0=0.1; y0=x0;

//discrétisation du plan tangeant sur une grille
dx=linspace(-1,1,10);dy=dx;
Zd=feval(dx,dy,fp);

//ajout du tracé du plan tangeant sur la surface et
//positionnement de l'angle de vue (paramètres alpha et theta)
plot3d(x0+dx,y0+dy,Zd,alpha=85,theta=50);
e2=gce();//l'objet associé à cette dernière surface
e2.color_mode=0;//facettes transparentes ou mesh
```

Fichier source : `scilab/plot3d_2.sce`

L'exécution de ces deux scripts produit le graphique de la figure 5.16.

Notons que, comme pour le tracé de courbe, le tracé de deux surfaces dans le même système d'axes génère automatiquement la composition de ces surfaces si la propriété `auto_clear` des axes est positionnée à `"off"`. Notons aussi que l'on aurait pu obtenir le même résultat en utilisant la variante `fplot3d` de la fonction

`plot3d` en remplaçant les instructions `Z=feval(x,y,f);` et `plot3d(x,y,Z)` par la instruction `fplot3d(x,y,f)`.

La fonction `plot3d` permet aussi de tracer des surfaces définies à partir de facettes. Une facette est un polygone dans l'espace défini par ses n sommets. Lorsque $n = 4$ la facette est quadrangulaire. Les coordonnées `x,y,z` de ces n points forment trois vecteurs colonnes. Un ensemble de N facettes donne donc trois matrices $n \times N$ `Xf,Yf,Zf`. La commande `plot3d(Xf,Yf,Zf)` dessine la surface constituée par ces N facettes. La fonction `genfac3d` peut être utilisée pour convertir la représentation précédente `(x,y,Z)` en facettes `(Xf,Yf,Zf)`.

En pratique, les surfaces sont souvent définies par une représentation paramétrique $x(u, v)$, $y(u, v)$, $z(u, v)$ où les paramètres (u, v) parcourent une partie du plan, souvent un rectangle. Pour dessiner de telles surfaces il peut être commode de les représenter par trois matrices de coordonnées X, Y, Z de même dimension et qui sont telles que 4 éléments voisins dans X, Y et Z sont les coordonnées d'une facette rectangulaire. La surface est alors tracée en parcourant toutes les sous-matrices 2×2 obtenues en sélectionnant 2 lignes et 2 colonnes contiguës dans X, Y et Z.

La fonction `plot3d2` dessine les surfaces ainsi définies par des facettes quadrangulaires et la fonction `nf3d` convertit la représentation par 3 matrices `(X,Y,Z)` correspondant à des facettes rectangulaires en représentation par facettes `(Xf,Yf,Zf)` utilisables par la fonction `plot3d`. Le script suivant montre comment ajouter une sphère au graphique précédent (figure 5.17).

```
//paramétrage en u et v
u = linspace(-%pi/2,%pi/2,20);  // 20 points de -pi/2 a +pi/2
v = linspace(-%pi,%pi,20);// 20 points de -pi a +pi
//(X Y Z) = Coordonnées de 20 x 20 points sur la sphère
R=0.3;//rayon de la sphère
c=[0 1.5 0];// centre de la sphère
X =cos(u)'*cos(v);  Y = cos(u)'*sin(v); Z = sin(u)'*ones(v);
// On dessine les quadrangles avec plot3d2 ...
plot3d2(c(1)+R*X,c(2)+R*Y,c(3)+R*Z,alpha=85,theta=50);
e3=gce();e3.color_mode=color('grey80');
a=gca();//a.isoview="on";a.cube_scaling="on";
```

Fichier source : `scilab/plot3d_3.sce`

Le graphique de la sphère aurait plus être aussi généré directement sous formes de facettes grâce à la fonction `eval3dp`.

Dans les exemples ci-dessus nous avons vu l'utilisation des arguments optionnels `theta` et `alpha` pour spécifier l'angle de vue. L'aide de la fonction `plot3d` présente de nombreux autres paramètres optionnels. Mais les graphiques 3D sont maintenant plus facilement configurables en accédant aux propriétés des objets graphiques.

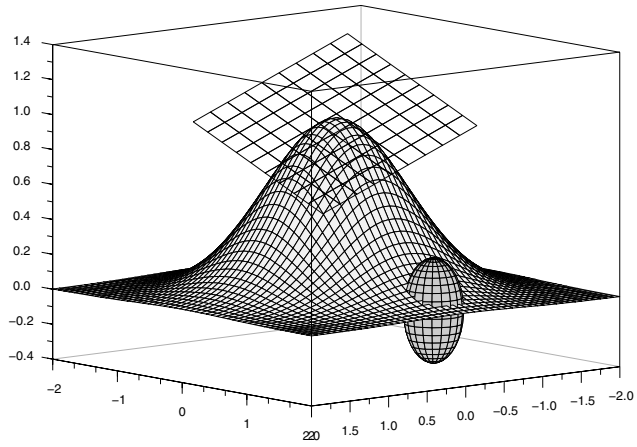


Figure 5.17 – Une sphère tracée avec `plot3d2`.

La fonction surf

La fonction `surf` est une émulation de la fonction Matlab du même nom ; tout comme `plot3d` et `plot3d1`, elle permet de visualiser les surfaces définies par les représentations `(x,y,Z)` et `(X,Y,Z)` présentées précédemment (voir page : 104). Ici, la surface est colorée en fonction de la cote moyenne de chacune des facettes.

Le script suivant donne un exemple de l'utilisation de la fonction `surf` pour visualiser la surface $z = 100\sin(x)\sin(y)e^{-x^2+2xy-y^2}$ et génère la figure 5.18.

```
function z=phi(x,y)
  z=100*sin(x).*sin(y).*(exp(-x.^2+2.*x.*y-y.^2));
endfunction
x=linspace(-2,2,40);y=x;
surf(x,y,feval(x,y,phi));
gcm=graycolormap(64);
gcm=gcm($-50:$,:);// suppression des couleurs les plus foncées
f=gcf();f.color_map=gcm;
```

Fichier source : `scilab/surf.sce`

5.4.3 Les objets associés aux surfaces

Il y a deux types d'objets graphiques permettant de représenter les surfaces : `Plot3d` pour les objets créés à partir de deux vecteurs et une matrice `(x,y,Z)` et `Fac3d` pour les représentations par facettes à partir de trois matrices `(Xf,Yf,Zf)`.

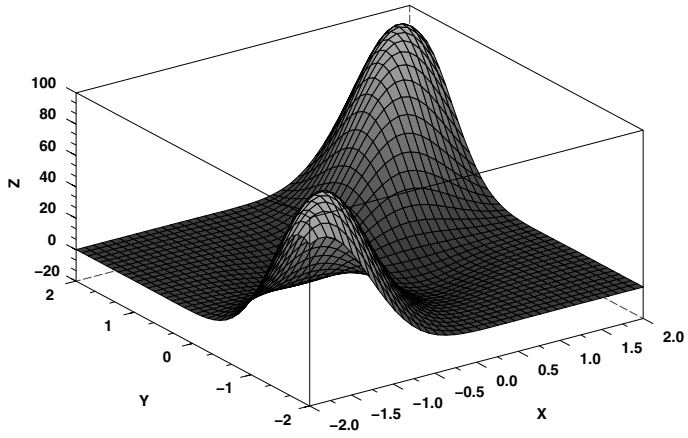


Figure 5.18 – Une surface tracée avec `surf`.

La visualisation des propriétés de l'objet associé à la surface en cloche de la figure 5.16 donne :

```
->e1
 e1  =
Handle of type "Plot3d" with properties:
========================================
parent: Axes
children: []
visible = "on"
surface_mode = "on"
foreground = -1
thickness = 1
mark_mode = "off"
mark_style = 0
mark_size_unit = "tabulated"
mark_size = 0
mark_foreground = -1
mark_background = -2
data.x = matrix 1x50
data.y = matrix 1x50
data.z = matrix 50x50
color_mode = 33
color_flag = 0
hiddencolor = 4
user_data = []
```

Le champ `data` est une `tlist` de type `3d` contenant les sous-champs `x` (vecteur), `y` (vecteur) et `z` (matrice). Pour l'objet `Fac3d` ce champ peut aussi contenir le sous-champ `color`

La propriété `hiddencolor` permet de spécifier la couleur de la partie cachée des facettes, une valeur positive correspond à un numéro de couleur, tandis qu'une valeur négative permet de colorier la partie cachée de la même couleur que la partie visible.

Les deux objets `Plot3d` et `Fac3d` sont assez similaires mais l'objet `Fac3d` est plus riche dans la mesure où il permet un meilleur contrôle de la colorisation des facettes. Cette colorisation est paramétrée par les propriétés `color_mode`, `color_flag` et `data.color` pour l'objet `Fac3d`. L'utilisation de ces propriétés est illustrée dans le script suivant qui produit la figure 5.19.

```
clf();f=gcf();f.color_map=graycolormap(16);
function myaxes()//fonction utilitaire
  a=gca();a.axes_visible='off';a.box="off";a.margins=[0 0 0 0];
  a.x_label.visible='off';
  a.y_label.visible='off';
  a.z_label.visible='off';
endfunction

t=[0:0.7:%pi]'; z=sin(t)*cos(t');[xx,yy,zz]=genfac3d(t,t,z);

subplot(231); //couleur uniforme
plot3d(xx,yy,zz);myaxes();
e=gce();e.color_flag=0; e.color_mode=10;

subplot(232); //colorisation selon z
plot3d(xx,yy,zz);myaxes();
e=gce();e.color_flag=1; e.color_mode=1;

subplot(233); //sans délimitation de facettes
plot3d(xx,yy,zz);myaxes()
e=gce();e.color_flag=1; e.color_mode=-1;
xtitle("color_flag=1; color_mode=-1")

subplot(234); //une couleur par facette
plot3d1(xx,yy,list(zz,16:-1:1));myaxes()
e=gce();e.color_flag=2;


subplot(235); //couleurs interpolées
//Matrices des couleurs aux points de la grille
C=[1;3;4;3;1]*ones(1,5)+ones(5,1)*(12:-3:0);
//couleurs aux  sommet des facettes
[xw,yw,cc]=genfac3d(t,t,C);
plot3d1(xx,yy,list(zz,cc));myaxes()
e=gce();e.color_flag=3;
```

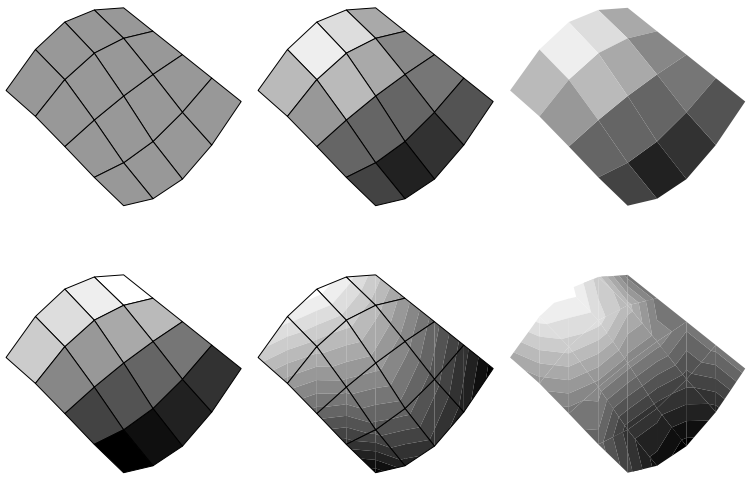


Figure 5.19 – Différentes représentations de la même surface.

```
subplot(236); //couleurs selon z interpolées
plot3d1(xx,yy,list(zz,zz));myaxes()
e=gce();e.color_flag=3; e.color_mode=-1;
e.cdata_mapping="scaled";
```

Fichier source : `scilab/surfaces.sce`

La fonction `myaxes` supprime la visualisation des axes et des légendes associés. Une description détaillée des propriétés des objets surface pourra être trouvée dans l'aide en ligne de `surface_properties`.

5.4.4 Autres visualisations de surfaces

Les fonctions `grayplot`, `Sgrayplot` et `contour2d` visualisent des représentations 2D des surfaces 3D, soit en utilisant la couleur pour visualiser la valeur de la cote `z`, soit en dessinant les lignes de niveaux. Le script suivant produit la figure 5.20 et montre une utilisation de ces deux fonctions pour une autre visualisation de la surface $z = 100 \sin(x) \sin(y) e^{-x^2+2xy-y^2}$ déjà présentée page 107.

```
clf();f=gcf();
//une table de gris adaptée
f=gcf();f.color_map=(1-linspace(0,1,64))'^2 *ones(1,3);

n=40;
```

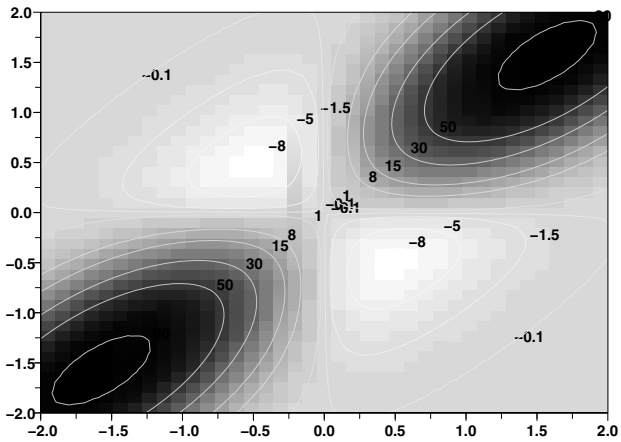


Figure 5.20 – Exemple des fonctions `grayplot` et `contour2d`.

```
x=linspace(-2,2,n);y=x;
//évaluation de la fonction sur une grille rectangulaire
Z=100* sin(x')*sin(y) .* ..
  (exp(-(x^2)'*ones(1,40)+2*x'*y-ones(n,1)*y^2));
grayplot(x,y,Z)

//superposition des courbes de niveau
l=[-8 -5 -1.5  -0.1 1  8 15 30 50 90];
contour2d(x,y,Z,l)
```

Fichier source : `scilab/grayplot.sce`

La fonction `phi` de la page 107 et l'appel à `feval` sont ici remplacés par une instruction vectorielle équivalente mais beaucoup rapide.

La fonction `contour2di` calcule les lignes de niveaux comme le montrent le script suivant et la figure 5.21 qu'il génère.

```
function p=Polyline3D(x,y,z);
  //crée et affiche une polyline 3D
  xpoly(x,y);p=gce(); p.data(:,3)=z(:);
endfunction

clf();f=gcf();
//une table de gris adaptée
f.color_map=(1-linspace(0,1,64))'^2 *ones(1,3);
```

```
n=40; x=linspace(-2,2,n); y=x;
Z=100* sin(x')*sin(y) .* ..
  (exp(-(x^2)'*ones(1,n)+2*x'*y-ones(n,1)*y^2));
surf(x,y,Z);
s=gce();s.color_mode=-1; //suppression des limites des facettes

//calcul des courbes de niveau
l=[-8 -5 -1.5  -0.1 1  8 15 30 50 90];
[xc,yc]=contour2di(x,y,Z,l);

//affichage des courbes de niveau sur la surface
k=1;n=yc(k);
drawlater(); //pour retarder l'affichage
while k+yc(k)<size(xc,'*')
  n=yc(k);
  p=Polyline3D(xc(k+(1:n)),yc(k+(1:n)),xc(k)+1);
  p.thickness=2;p.foreground=-2;
  k=k+n+1;
end
drawnow(); // afficher les graphiques
```

Fichier source : `scilab/contour2di.sce`

On notera l'utilisation de la fonction `drawlater` qui permet de différer l'affichage des objets graphiques (par défaut l'ensemble des objets d'une figure est redessiné lorsque l'on change une propriété de l'un d'entre eux) et l'utilisation de la fonction `drawnow` qui permet le retour au mode par défaut et provoque l'affichage. L'utilisation de ces deux fonctions permet d'éviter le clignotement de l'affichage lorsqu'un graphique résulte de la création et de la modification de plusieurs objets.

5.4.5 Textes et légendes

On peut faire figurer des légendes sur un graphique en utilisant la fonction `legend`. Les légendes apparaissent dans un petit rectangle, dont la position peut être contrôlée, avec un trait de rappel qui correspond au type de ligne des objets `Polyline` correspondants. Les fonctions `xstring` et `xstringb` permettent aussi d'écrire du texte dans un graphique. Ainsi le script suivant produit la figure 5.22.

```
x=linspace(0,5,40)';
plot2d(sin(x),[cos(2.5*x) cos(1.2*x)])
e=gce();
p1=e.children(1); //cos(1.2*x)
p2=e.children(2); //cos(2.5*x)
p1.polyline_style=4;   //arrow mode
p1.arrow_size_factor=2;
p2.polyline_style=4;
```

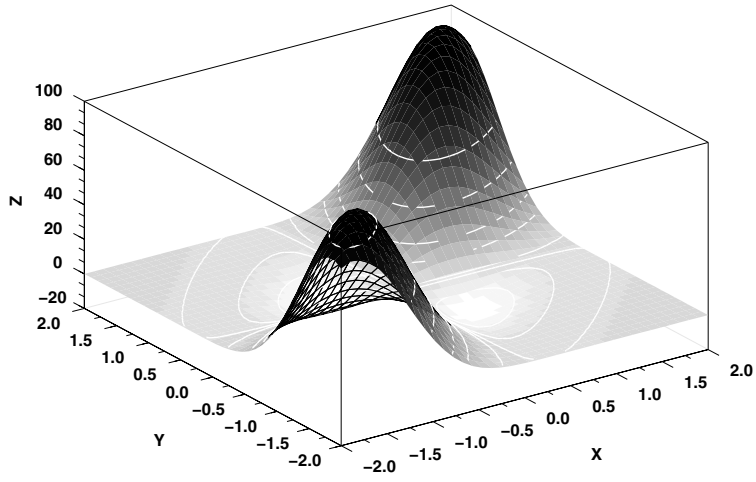


Figure 5.21 – Utilisation de la fonction `contour2di`.

```
p2.arrow_size_factor=2;
p2.line_style=3; //dash
legend(['sin(x) vs cos(2.5x)';'sin(x) vs cos(1.2x)']);
xstring(-0.9,1,['Une courbe de Lissajous peut être définie par'
  'x(t)=a sin(t), y(t)=b sin(n t +c)'])
```

Fichier source : `scilab/lissajou.sce`

5.5 Autres fonctions graphiques

Il existe de nombreuses autres fonctions graphiques, on peut les lister par la commande `help Graphics`. On peut ainsi tracer des segments, des arcs, des flèches (fonctions `xsegs`, `xarcs`, `xarrows`), dessiner et remplir des rectangles, des polygones, (fonctions `xrects`, `xfrect`, `xpolys`, `xfpolys`), placer des chaînes de caractères (fonction `xstring`). Ces fonctions agissent dans le système de coordonnées (`Axes`) mais, contrairement aux fonctions de haut niveau vues précédemment, n'agissent pas sur ses paramètres. Il peut donc être nécessaire d'initialiser correctement les paramètres des axes, notamment l'échelle.

Dans l'exemple suivant on dessine 7 rectangles de couleur (un par jour de la semaine). La hauteur de chaque rectangle est donnée et la couleur est prise dans une table de couleurs pré-définie (`hotcolormap`). On utilise la fonction

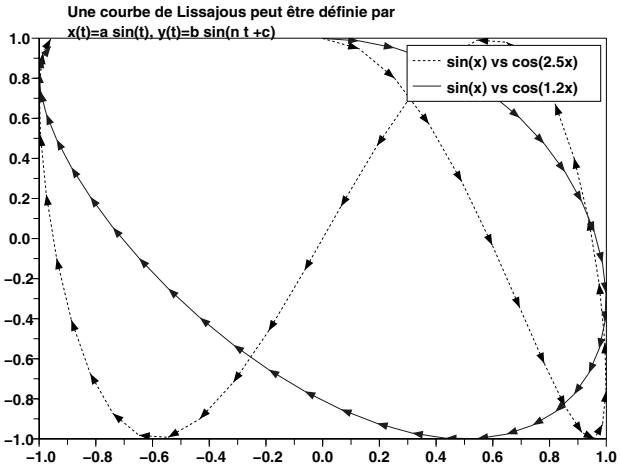


Figure 5.22 – Courbes de Lissajous et `legend`.

`xrects` pour tracer les rectangles et la fonction `xstring` pour placer les chaînes de caractères, Le script génère la figure 5.23.

```
Jours=["Lundi"," Mardi","Mercredi",..
       "Jeudi","Vendredi","Samedi","Dimanche"];
Njours=size(Jours,2);  // 7
V=[5,9,6,12,10,8,4];  //Hauteurs des rectangles
//On definit une table de couleurs
f=gcf();f.color_map=graycolormap(3*Njours);
//Un cadre de taille 7 sur 14 (Njours+1,max(V)+1)
a=gca();a.data_bounds=[0,0;Njours,max(V)+1];

//On trace 7 rectangles de hauteur V
xleft=0:Njours-1;yupper=V;largeurs=ones(1,Njours);hauteurs=V;
xrects([xleft;yupper;largeurs;hauteurs],3*Njours-V)
//On précise leurs propriétés
e=gce();e.children(1:$).thickness=2;
e.children(1:$).line_mode="on";

//Paramétrage des axes
a.auto_ticks='off';
a.axes_visible = ["on","off","off"];
xticks=a.x_ticks;
xticks.locations=0.5+(0:6); xticks.labels=Jours;
a.x_ticks=xticks;
a.sub_ticks = [0,0];
```

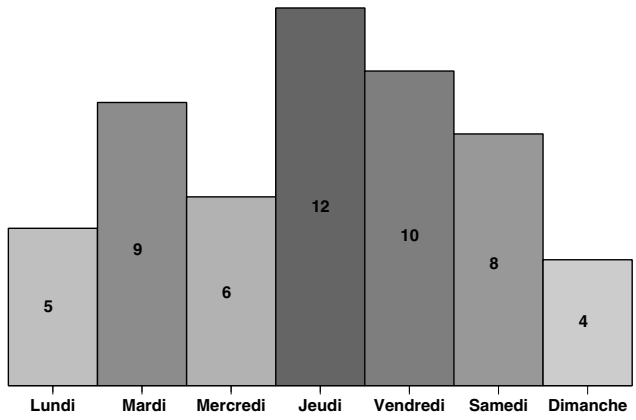


Figure 5.23 – Des rectangles de couleur.

```
// on écrit la valeur dans chaque rectangle
for k=1:Njours; xstring(k-1+0.4,V(k)/2.2,string(V(k))); end
```

Fichier source : `scilab/semaine.sce`

D'autres fonctions sont dédiées à des tâches spécifiques telles que le tracé d'un champ de vecteurs (fonction `champ`, `fchamp`). Enfin la fonction `Matplot` permet d'afficher des images.

5.6 Interaction avec la fenêtre graphique

5.6.1 Fonction xclick

La fonction `xclick` s'utilise lorsque l'on veut récupérer les coordonnées d'un point de la fenêtre graphique. La fonction `xclick` est bloquante ; elle attend un événement souris, menu ou clavier. Dans l'exemple suivant, on illustre comment on peut afficher une table de couleurs et récupérer un numéro de couleur par un clic dans une fenêtre graphique. Cette fonction est une version simplifiée de la fonction `getcolor`. On définit une fenêtre graphique sans menu et sans marge, remplie de quatre rectangles de couleur numérotés de 1 à 4. Après un clic dans l'un des rectangles, le numéro du rectangle choisi est affiché dans la zone de dialogue de la fenêtre graphique.

```
function k=getcouleur()  // fichier getcouleur.sci
//On ouvre une nouvelle fenêtre graphique
  f=scf(max(winsid()+1));

  // Paramétrage de la fenêtre
  f.figure_size=[100,100];// taille en pixels
  //On definit une table à 4 couleurs
  f.color_map=[1 0 0;//rouge
       0 1 0; //vert
       0 0 1; //bleu
       1/2 1/2 1/2; //gris
      ];
  Ncolors=4;

  // Paramétrage des Axes
  a=gca();a.margins=[0 0 0 0]; //pas de marges

  //On trace 4 rectangles remplis des couleurs de la table
  h=1/2;
  rect_rouge=[0 1 h h]';rect_vert=[0 1-h h h]';
  rect_bleu=[h 1 h h]';rect_gris=[h 1-h h h]';
  rects=[rect_rouge rect_vert rect_bleu rect_gris];
  xrects(rects,1:Ncolors);

  for k=1:Ncolors   //On numérote chacun des quatre carrés
    xstringb(rects(1,k),rects(2,k)-h,string(k),h,h,"fill")
  end

  k=[];
  while %t //boucle d'interaction
    [c_i,cx,cy]=xclick();//(cx,cy): coordonnées du point
    if or(c_i==[2 5 -100])  then
      //bouton droit pressé ou cliqué ou fenêtre fermée
      break,
    end
    c=find(cx>=[0 h]&cx<[h 2*h]) // numéro de colonne
    l=find(cy<[1 1-h]&cy>=[1-h 1-2*h]) //numéro de ligne
    k=2*(c-1)+l; //couleur selectionnée
    //Un titre dans le fenêtre graphique
    xinfo("Vous avez choisi la couleur numéro: "+string(k))
  end

  if c_i<>-100 then delete(f);end  //destruction de la fenêtre
endfunction
```

Fichier source : `scilab/getcouleur.sci`

La variable de retour `c_i` contient un indicateur de la source et du type d'événement associé à la sélection du point. Les valeurs 0, 1 ,2 correspondent

à la détection d'une action « bouton pressé » pour les boutons gauche, milieu et droit de la souris ; les valeurs 3, 4, 5 correspondent à la détection d'un clic de ces mêmes boutons. La valeur -100 signifie que la fenêtre a été fermée par l'utilisateur. Se reporter à l'aide en ligne de `xclick` pour plus de détails sur la signification de cet indicateur.

La fonction `locate` construite sur `xclick` permet de récupérer un ensemble de points.

5.6.2 Fonction xgetmouse

La fonction `xgetmouse` permet de connaître la position de la souris dans la fenêtre lorsqu'un événement, y compris un déplacement de la souris, se produit. Dans l'exemple qui suit, une marque est tracée aux différentes positions de la souris :

```
// spécification de la fenêtre et des axes
f=gcf();clf();
a=gca();a.data_bounds=[0 0;100 100];
a.margins=[0 0 0 0];

// création d'une marque
xpoly(50,50,"marks");
e=gce();e.mark_style=3;

while %t //boucle d'interrogation
  r=xgetmouse();
  if or(r(3)==[2 5 -100]) then
    //le bouton droit a été pressé ou cliqué ou
    //la  fenêtre a été fermée
    break;
  elseif r(3)==-1 then //déplacement
     e.data=r(1:2);
  elseif or(r(3)==[0 3]) then //clic gauche
    e=copy(e); //copie de la marque
  elseif or(r(3)==[1 4]) then //clic milieu
     e.mark_style= modulo(e.mark_style,10)+1;
  end
end
```

Fichier source : `scilab/xgetmouse.sce`

`r(1:2)` contient les coordonnées utilisateurs de la position de la souris, `r(3)` contient l'indicateur de l'événement. La valeur -1 correspond à un mouvement de la souris, tandis que les valeurs 0 à 5 et -100 ont la même signification que le paramètre de retour `c_i` de la fonction `xclick`. Pour plus de détails sur cette fonction, on pourra se référer à son aide en ligne.

5.6.3 Gestionnaire d'événements

Les fonctions précédentes attendent qu'un événement se produise ou traitent un événement précédent qui a été mémorisé. Ainsi, l'exécution du code est synchronisée avec les événements.

Le mécanisme de `event handler` permet de réagir à des évènements à tout moment. La fonction `seteventhandler` permet de définir la fonction à appeler en réaction à un événement se produisant dans la fenêtre graphique correspondante. La fonction ci-dessous définit un tel gestionnaire d'événements. La liste d'appel de cette fonction est imposée.

```
function my_eventhandler(win,x,y,ibut)
  if ibut==-1000 then return,end //la fenêtre a été fermée
  [x,y]=xchange(x,y,'i2f') //pixel vers coordonnées utilisateur
  a=gca(); e=a.children(1);

  if ibut==-1 then // déplacement
    e.data=[x,y];
  elseif or(ibut==[0 3]) then //clic gauche
    copy(e) //copie de la marque
  elseif or(ibut==[1 4]) then //clic milieu
    //changement de symbole
    e.mark_style= modulo(e.mark_style,10)+1;
  elseif or(ibut==[2 5]) then //clic droit
    seteventhandler("") // suppression du handler d'évenements
  end
endfunction
```

Fichier source : `scilab/my_eventhandler.sci`

Le gestionnaire d'événements ci-dessous permet d'animer le symbole créé par le script :

```
f=gcf();clf();
a=gca();a.data_bounds=[0 0;100 100];
a.margins=[0 0 0 0];
// création d'une marque
xpoly(50,50,"marks");
e=gce();e.mark_style=3;

//association du gestionnaire d'évenements à cette fenêtre
seteventhandler("my_eventhandler")
```

Fichier source : `scilab/eventhandler.sce`

Noter l'absence de boucle dans ce script : le script se termine donc après avoir mis en place le gestionnaire d'événements de la fenêtre mais ce gestionnaire continue de réagir aux événements de la fenêtre à laquelle il est associé, tant qu'il n'est pas supprimé par l'instruction `seteventhandler("")` ou par la destruction de la fenêtre.

5.7 Animation

Pour réaliser une animation il suffit d'exécuter un script qui réalise chacune des vues en séquence. Cependant, si l'on ne prend pas de précautions, le rendu peut être désagréable : scintillement, lenteur, ... Nous présentons ici les outils graphiques permettant d'avoir un bon rendu.

5.7.1 Fonctions drawlater et drawnow

Comme évoqué en section 5.4.4, par défaut, la modification d'un objet graphique provoque le réaffichage de l'ensemble des objets contenus dans la fenêtre graphique. Ce mode par défaut provoque un affichage qui « saute » lorsque l'on construit un graphique nécessitant beaucoup de manipulations. Dans de tels cas la fonction `drawlater` permet de différer les affichages : les objets sont créés ou modifiés en mémoire mais ne sont pas affichés. L'appel à la fonction `drawnow` provoque le dessin des objets et repositionne le mode par défaut. L'utilisation de ces deux fonctions permet de diminuer le clignotement de l'affichage et de réduire sensiblement le temps de calcul.

La fonction `draw` peut être utilisée pour provoquer le dessin d'un objet spécifié par son handle ainsi que de toute sa descendance.

5.7.2 Mode « double tampon »

Alors que par défaut les ordres d'affichage sont directement envoyés à l'écran, sous le mode « double tampon » (que l'on active avec l'instruction `set(gcf(), "pixmap","on")`) les dessins sont réalisés d'abord en mémoire. Ainsi, la fonction `show_pixmap` permet de remplacer les pixels de l'écran par ceux de la mémoire interne, sans effacement préalable de l'écran. Il est ainsi possible de masquer les transitions d'une vue à l'autre et de réaliser des animations fluides.

L'exemple ci-dessous permet d'afficher l'évolution temporelle de la fonction $y = x\sin(x+t)$.

```
clf();f=gcf();
f.pixmap='on'; //mode double tampon

//bornes du graphique
a=gca();
a.data_bounds=[0 -6;2*%pi 6];

//vue initiale
x=linspace(0,2*%pi,100)';
plot2d(x,x.*sin(x));
e=gce();e=e.children; //le handle sur la courbe

for t=linspace(0,20,500) //boucle d'animation
  e.data(:,2)=x.*sin(x+t);
  show_pixmap();
```

```
end
f.pixmap='off';
```

Fichier source : `scilab/animation.sce`

La fonction `xpause` qui bloque l'exécution pendant un laps de temps donné et la fonction `realtime` qui attend qu'un instant donné soit atteint permettent de ralentir, si nécessaire, l'affichage.

5.7.3 Mode xor

Dans les cas les plus complexes, il peut être nécessaire de recourir à l'utilisation du mode `xor`. Ce mode que l'on spécifie par la propriété `pixel_drawing_mode` des objets `Figure` régit la façon dont un pixel est affiché à l'écran. Dans le mode par défaut (`copy`) le pixel est affiché à l'écran dans la couleur requise. Plus généralement cette propriété permet de spécifier un opérateur logique entre les bits du pixel déja présent sur l'écran et ceux du pixel à afficher.

L'opérateur logique `xor` vérifie `xor(xor(a,b),a)=b`. Sous ce mode, dessiner une seconde fois un objet donné l'efface. Cette astuce peut permettre d'animer de façon efficace les évolutions d'objets graphiques sur une scène fixe et chère à réafficher. Il y cependant une contrepartie : les couleurs affichées ne sont pas forcément celles que l'on désire.

Dans le script qui suit on affiche deux rectangles colorés en noir en utilisant la fonction `xfrect`. La couleur est donnée par la variable d'environnement `ei.background` où `ei` est l'entité courante. Le premier rectangle traverse la fenêtre graphique de bas en haut et de gauche à droite et le second traverse la fenêtre graphique de haut en bas et de droite à gauche. On voit que dans le mode `xor` la couleur de l'intersection est la même que la couleur du fond (voir la figure 5.24).

```
clf();f=gcf();
f.pixmap='on';   //double buffer mode
f.pixel_drawing_mode='xor';   //mode xor

ax=gca();
ax.data_bounds=[0,-4;14,10];//on fixe les bornes
ax.margins=[0 0 0 0];
ax.background=color("lightgrey");
MAX=10;
//on crée les objets rectangles dans leur position initiale
k=1;
//on trace un rectangle noir en k,k
xfrect(k,k,4,4); e1=gce();
//on trace un rectangle noir en MAX-k,MAX-k
xfrect(MAX-k,MAX-k,4,4);e2=gce();
```

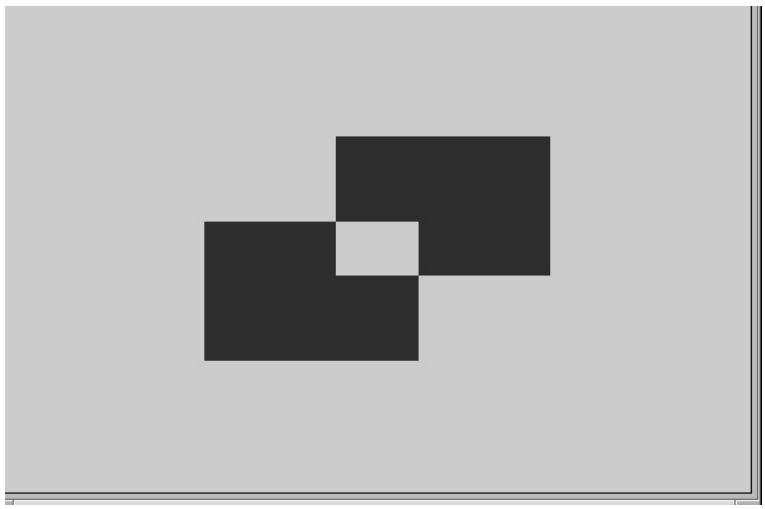


Figure 5.24 – Illustration du mode `xor`.

```
//boucle de déplacement
for k=linspace(1,10,1000)
  e1.data(1:2)=k;
  e2.data(1:2)=MAX-k;
  show_pixmap() //affichage double buffer
end
```

Fichier source : `scilab/xor.sce`

5.8 Exportation des figures

À l'aide du menu `Fichier`, on peut imprimer le contenu de la fenêtre graphique sélectionnée[3].

On peut aussi exporter le graphique Scilab dans des fichiers aux formats PostScript, Latex, Xfig, Gif, PPM ou EnhMetafile en sélectionnant le menu `Fichier/Exporter` puis le format et l'orientation désirés. On peut aussi le faire en utilisant directement les commandes `xs2eps`, `xs2emf`, `xs2gif`, `xs2ppm` ou `xs2fig` pour exportation vers Postscript, EnhMetafile, Gif, PPM ou Xfig respectivement.

Les formats PostScript, Xfig et EnhMetafile sont des formats vectoriels qui produisent des sorties de qualité et qu'il est possible de zoomer tandis que les formats Gif et PPM sont des formats bitmap.

[3]Sous Unix, les imprimantes proposées sont celles définies par la variable d'environnement PRINTER.

Pour inclure un graphique dans un document, le plus simple est de l'exporter dans un format adapté. On peut par exemple inclure un graphique dans un document Latex en l'exportant au format PostScript dans le fichier `foo.eps` puis en utilisant les commandes du package LaTeX `epsfig` :

```
\begin{center}
\includegraphics[width=12cm]{foo.eps}
\end{center}
```

Sous Windows on pourra exporter le graphique au format `EnhMetafile` qui est aussi un format vectoriel pour l'inclure, par exemple, dans Word.

Pour inclure un graphique Scilab dans une page Web on préférera exporter au format Gif ou PPM, des outils externes comme `xv`, `Gimp`, `ImageMagic` permettant alors de générer des fichier aux formats JPEG, ...

On peut aussi exporter des figures par les commandes `xs2eps`, `xs2emf`, `xs2gif`, `xs2ppm` ou `xs2fig` pour exportation vers Postscript, EnhMetafile, Gif, PPM ou Xfig respectivement.

La largeur et la hauteur de la fenêtre graphique sont données, en pixels, par les instructions `f=gcf(); f.axes_size` et valent par défaut 600 pixels sur 400. Évidemment, ces valeurs sont changées chaque fois que l'on redimensionne la fenêtre à la souris.

Lorsque le contenu d'une fenêtre est exporté, on change automatiquement de pilote, par exemple on passe au pilote PostScript (`driver("Pos")`). On peut aussi positionner directement le pilote par la fonction `driver` et ouvrir un fichier par la commande `xinit`, puis exécuter les ordres graphiques désirés, et enfin fermer le fichier d'export avec la fonction `xend`.

5.9 Sauvegarde et rechargement des entités graphiques

Il est possible de sauvegarder les entités graphiques sur fichier binaire dans un format propre à Scilab et de les recharger ensuite. Le menu `Fichier/Sauvegarder` ou `File/Save` de la fenêtre graphique permet de sauver la totalité de la figure associée. Le rechargement s'effectue par le menu `Fichier/Charger` ou `File/Load`.

Ces mêmes opérations sont aussi réalisables en utilisant les fonctions `xsave` et `xload` : `xsave("mon_graphique.scg",0)` sauve le contenu de la fenêtre graphique numéro `0` dans le fichier binaire `mon_graphique.scg` tandis que l'instruction `xload("mon_graphique.scg")` recharge ce même fichier dans la fenêtre graphique courante sans l'effacer au préalable.

La fonction `save` appliquée à une variable de type `handle` sauve l'ensemble de la définition de l'objet graphique associé dans le fichier binaire spécifié. Le rechargement d'une telle variable recrée l'objet dans la fenêtre graphique courante ainsi que la variable de type handle dans l'environnement Scilab :

```
clf() //efface la fenêtre courante
```

```
a=gca();
a.data_bounds=[0 0;10 10];
xrect(1,1,2,2); //un rectangle
r=gce(); //le handle sur le rectangle
r.thickness=2; //épaisseur du trait
r.background=color('red'); //couleur de remplissage
r.fill_mode = "on"; //mode remplissage
save('rect.scg',r) //sauvegarde sur fichier
delete(r);//destruction du rectangle
load('rect.scg') //rechargement
```

Fichier source : `scilab/saveloadhandle.sce`

Chapitre 6

Programmation avancée

6.1 Structures dans Scilab

Nous avons déjà évoqué les objets de base de Scilab (matrices de nombres, de polynômes, de chaînes de caractères, de booléens, ...). Mais pour développer des programmes complexes avec un code lisible et concis, le programmeur a besoin de structures pouvant contenir des données hétérogènes. Scilab permet de définir et de manipuler ces nouveaux objets aisément.

6.1.1 list

On a déjà vu la création des `list` (voir paragraphe 2.2.5). L'instruction :

```
->tableau=list(["x";"y"],["a","b","c"],rand(2,3));
```

crée une structure de données représentant un tableau de nombres auquel sont associés des noms de lignes et de colonnes. Les différents champs de cette structure sont accessibles par leur rang dans la structure.

```
->tableau(2)

 ans =

    !a b c !

->sum(tableau(3)(1,:))
 ans  =

    1.1943134
```

Il est bien évidemment possible de changer la valeur d'un champ :

```
->tableau(3)=[1 2 3;4 5 6];

->tableau(3)
 ans  =

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

voire même d'une sous-matrice contenue dans l'un des champs :

```
->tableau(3)(:,2)=[0;0];

->tableau(3)
 ans  =

!   1.    0.    3. !
!   4.    0.    6. !
```

Ce type de structure peut être récursif, définissant ainsi des structures de données arborescentes, comme dans l'exemple ci-dessous où l'on veut décrire l'arbre des descendants d'une personne. Un individu y est décrit par une structure contenant son état civil (prénom, date de naissance, liste des enfants). Cette structure est créée par la fonction `etat_civil` :

```
function individu=etat_civil(prenom,date_naissance,enfants)
  if argn(2)==2 then enfants=list(),end
  individu=list(prenom,date_naissance,enfants)
endfunction
```

Fichier source : `scilab/etat_civil.sci`

On retrouve la fonction `argn` qui permet de gérer un nombre variable d'arguments.

Exemple d'utilisation

```
->JeanPierre=etat_civil("Jean-Pierre",1908)
 JeanPierre  =

       JeanPierre(1)

 Jean-Pierre

       JeanPierre(2)
```

```
    1908.

       JeanPierre(3)

     ()
```

On observe ci-dessus la visualisation par défaut des structures.

La fonction `nouveau` ajoute un nouveau descendant à un ascendant spécifié par son prénom.

```
function l=nouveau(l,individu,ascendant)
  if l(1)==ascendant then // L'ascendant désigné a été trouvé
    l(3)($+1)=individu    // Extension de la liste des enfants
  else
    // Parcours de l'arbre des descendants
    for k=1:size(l(3))
      l(3)(k)=nouveau(l(3)(k),individu,ascendant)
    end
  end
endfunction
```

Fichier source : `scilab/nouveau.sci`

L'appel récursif de la fonction `nouveau` permet d'effectuer simplement le parcours de cette structure arborescente.

Pour illustrer le fonctionnement de ces fonctions, créons la descendance de Jean-Pierre :

```
arbre=etat_civil("Jean-Pierre",1908);// l'ancêtre
// et sa descendance
arbre=nouveau(arbre,etat_civil("Maurice",1928),"Jean-Pierre");
arbre=nouveau(arbre,etat_civil("Francois",1930),"Jean-Pierre");
arbre=nouveau(arbre,etat_civil("Marianne",1953),"Francois");
arbre=nouveau(arbre,etat_civil("Claude",1960),"Maurice");
arbre=nouveau(arbre,etat_civil("Serge",1965),"Maurice");
arbre=nouveau(arbre,etat_civil("Jean-Philippe",1985),"Marianne");
```

Fichier source : `scilab/arbre.sce`

La visualisation par défaut de la structure `arbre` n'est pas très lisible. Nous définissons donc la fonction récursive `affiche_arbre` pour réaliser une visualisation spécifique.

```
function affiche_arbre(descendants,shift)
  //indentation initiale
  if ~exists("shift","local") then shift="",end
```

```
  // affiche le niveau courant
  mprintf("%s%s %d\n",shift,descendants(1),descendants(2))
  shift=shift+"     ";// augmentation de l'indentation

  // boucle sur les descendants
  for k=1:size(descendants(3))
    // appel recursif pour  les descendants
    affiche_arbre(descendants(3)(k),shift)
  end
endfunction
```

Fichier source : `scilab/affiche_arbre.sci`

On notera l'usage de la fonction `exists` avec l'option `"local"` pour tester si l'argument `shift` a été fourni. Il aurait aussi été possible d'utiliser la fonction `argn`.
Si on appelle cette fonction avec l'arbre créé précédemment on obtient :

```
->affiche_arbre(arbre)

Jean-Pierre 1908
     Maurice 1928
          Claude 1960
          Serge 1965
     Francois 1930
          Marianne 1953
               Jean-Philippe 1985
```

6.1.2 tlist

Le programmeur veut quelquefois désigner les champs par des noms symboliques plutôt que par des numéros. Cela peut être fait en utilisant les structures « typées » créées par la fonction `tlist`. Ce type de structure hérite de toutes les propriétés des structures précédentes mais il est en outre possible d'y attacher un type symbolique ainsi que de nommer les champs. La fonction `etat_civil` peut être modifiée pour utiliser ce type de structure :

```
function individu=etat_civil(prenom,date_naissance,enfants)
  if argn(2)==2 then
    enfants=tlist(["etat","prenom","date","enfants"]);
  end
  individu=tlist(["etat","prenom","date","enfants"],..
 prenom,date_naissance,enfants)
endfunction
```

Fichier source : `scilab/etat_civil1.sci`

Le premier champ de la nouvelle structure `individu` est un vecteur de chaînes de caractères. La première chaîne donne le type symbolique de la structure et les suivantes les noms symboliques des champs. La fonction `nouveau` s'écrit alors de façon plus lisible :

```
function l=nouveau(l,individu,ascendant)
  if l.prenom==ascendant then//L'ascendant désigné a été trouvé
    l.enfants($+1)=individu  //Ajout de l'enfant
  else
    for k=1:size(l.enfants)  //Parcours de l'arbre
      l.enfants(k)=nouveau(l.enfants(k),individu,ascendant)
    end
  end
endfunction
```

Fichier source : `scilab/nouveau1.sci`

Le premier champ de la structure `tlist` contient un vecteur de chaînes de caractères dont le premier est le nom du type et les suivants les noms des champs de la structure. On notera la notation "." permettant d'accéder aux champs par leurs noms. L'indexation `l.enfants` est équivalente à `l("enfants")` ou encore à `l(4)`.

La fonction `%etat_p` ci-dessous, adaptée de `affiche_arbre`, permet de surcharger la visualisation par défaut des structures de type `etat`. Le nom de la fonction (`%etat_p`) est construit à partir du nom symbolique du type `etat` et du symbole du display `p`. Pour plus de détail sur la surcharge des opérateurs et des fonctions, voir la partie 6.2 ainsi que l'aide en ligne sur `overloading`.

```
function %etat_p(descendants,shift)
  if ~exists("shift","local") then shift="",end
  // affiche le niveau courant
  mprintf(shift+"%s %i",descendants.prenom,descendants.date)
  // accroit le décalage les descendants
  shift=shift+"     ";// décalage du texte relatif à ce niveau

  // boucle sur les descendants
  for k=1:size(descendants.enfants)
    // appel récursif  pour traiter les descendants
    %etat_p(descendants.enfants(k),shift)
  end
endfunction
```

Fichier source : `scilab/affiche_arbre1.sci`

On note ici l'usage de la fonction `mprintf` décrite au paragraphe 4.2.2. Il n'est alors plus nécessaire d'appeler explicitement une fonction pour réaliser la visualisation de la structure `arbre` :

```
->arbre

Jean-Pierre 1908
     Maurice 1928
          Claude 1960
          Serge 1965
     Francois 1930
          Marianne 1953
               Jean-Philippe 1985
```

De la même façon, la sémantique de l'insertion peut être surchargée en définissant la fonction `%etat_i_etat` comme ci-dessous. Le nom de la fonction est construit à partir du nom symbolique des opérateurs (ici `etat`) et du symbole de l'opérateur (ici « `i` » pour l'insertion).

```
function l=%etat_i_etat(ascendant,individu,l)
  if l.prenom==ascendant then
    l.enfants($+1)=individu
  else
    for k=1:size(l.enfants)  // boucle sur les descendants
      l.enfants(k)=nouveau(l.enfants(k),individu,ascendant)
    end
  end
endfunction
```

Fichier source : `scilab/etati.sci`

La définition de la descendance se fait alors plus simplement :

```
arbre=etat_civil("Jean-Pierre",1908);// l'ancetre
// et sa descendance
arbre("Jean-Pierre")=etat_civil("Maurice",1928);
arbre("Jean-Pierre")=etat_civil("Francois",1930);
arbre.Maurice=etat_civil("Claude",1960);
arbre.Maurice=etat_civil("Serge",1965)
arbre.Francois=etat_civil("Marianne",1953)
arbre.Marianne=etat_civil("Jean-Philippe",1985)
```

Fichier source : `scilab/arbre1.sce`

Ici on a utilisé la syntaxe `arbre("Jean-Pierre")` au lieu de la notation « `.` » à cause du symbole opératoire « `-` » présent dans le prénom.

6.1.3 mlist

La structure de type `mlist` est très similaire à la structure `tlist` mais elle permet à l'utilisateur de définir ses propres algorithmes d'indexation. Il est

ainsi possible de créer des structures que l'on souhaite interpréter comme des tableaux. A titre d'exemple examinons les structures et les fonctions utilisées dans Scilab pour manipuler les tableaux Excel. La fonction `readxls` permet de lire les fichiers Excel. Le résultat de la lecture du fichier ainsi que le contenu des différentes feuilles sont stockés dans des structures de type `mlist`. Une feuille est stockées dans une `mlist` qui a la structure suivante :

```
feuille=mlist(['xlssheet','name','text','value'],..
                          nom_feuille,Chaines,Valeurs)
```

Le champ `text` contient une matrice de chaînes de caractères dont certaines entrées peuvent être des chaînes vides, et le champ `value` contient une matrice de nombres, certains élément pouvant avoir la "valeur" `NaN` (not a number).

La fonction `%xlssheet_p` ci-dessous permet de définir la visualisation de ce type de structure

```
function %xlssheet_p(sheet)
  //visualise une feuille
  T=sheet.text
  V=sheet.value
  k=find(~isnan(V))
  T(k)=string(V(k))
  disp(T)
endfunction
```

Fichier source : `scilab/xlssheet_p.sci`

La fonction `%xlssheet_e` ci-dessous permet de définir les syntaxes d'extraction :
`feuille(i)` ou `feuille(i,j)`

```
function R=%xlssheet_e(varargin)
  //extrait une sous matrice d'une feuille
  s=varargin($) //La structure source
  R=s.value;T=s.text
  R=R(varargin(1:$-1)) //R(i) ou R(i,j)
  T=T(varargin(1:$-1)) //T(i) ou T(i,j)
  if and(isnan(R)) then //Il n'y a que du texte
    R=T //retour: une matrice de chînes
  elseif or(isnan(R)) then
    R=mlist(["xlssheet","name","text","value"],"",T,R)
  end
endfunction
```

Fichier source : `scilab/xlssheet_e.sci`

Les instructions suivantes montrent un exemple d'utilisation de ces structures.

```
->Sheets = readxls('SCI/demos/excel/t1.xls');
->feuille1 = Sheets(1) // visualisation par %xlssheet_p
 ans  =

!toto  tata  !
!            !
!1     2     !

//indexation par %xlssheet_e
->feuille1(4)
 ans  =

    2.
->feuille1(:,1)
 ans  =

!toto  !
!      !
!1     !
```

6.1.4 cell

Supposons que le programmeur veuille définir des tableaux multidimensionnels dont les éléments sont de n'importe quel type. Cela peut se faire en Scilab en utilisant les structures de type `cell`. Ces structures qui sont un cas particulier des `mlist` (de type `ce`) émulent les objets `cell` de Matlab.

La fonction `makecell` permet de construire de tels tableaux à partir du vecteur des dimensions et de la séquence des éléments. Ces éléments seront rangés dans le tableau colonne par colonne :

```
 ->M=makecell([2,2],"foo",[1 2
3],[4;5],list(1,2,3))
 M  =


!"foo"  [1,2,3]      !
!                     !
![4;5]  list(1,2,3)  !
```

Ces objets `cell` peuvent être manipulés comme des matrices normales :

```
->M(1,:)
 ans  =

!"foo"  [1,2,3]  !
->M(2,1)=M(1,2)
```

```
 M  =

!"foo"    [1,2,3]      !
!                      !
![1,2,3]  list(1,2,3)  !

-> [M M]
 ans  =

!"foo"    [1,2,3]      "foo"    [1,2,3]      !
!                                            !
![1,2,3]  list(1,2,3)  [1,2,3]  list(1,2,3)  !
```

Il faut toutefois noter que l'indexation d'élément au sein d'une `cell` requiert une syntaxe particulière :

```
->typeof(M(1,1)) //M(1,1) est une cell
 ans  =

 ce
 ->typeof(M(1,1).entries)
 ans  =

 string

->M(1,1).entries=1+%s
 M  =


!1+s      [1,2,3]      !
!                      !
![1,2,3]  list(1,2,3)  !
```

L'indexation classique `M(i,j)` référence la sous-`cell`.

6.1.5 struct

Le type `struct` est aussi un cas particulier de `mlist` (type `st`) qui émule le comportement des `struct` de Matlab. Sa principale spécificité est d'accepter l'ajout de nouveau champ lorsque l'on insère un champ inexistant :

```
->clear S;
->S.a=1:3// struct avec un seul champ a.
 S  =

   a: [1,2,3]

->S.b="Scilab" //ajout du champ b
```

```
 S  =

   a: [1,2,3]
   b: "Scilab"
```

L'objet `struct` supporte aussi un adressage de type tableau multidimensionnel :

```
->S(1,2).a=%pi
 S  =

1x2 struct array with fields:
   a
   b
```

6.2 Surcharge des opérateurs et des fonctions

Quelques exemples de surcharge d'opérateurs ont été présentés au paragraphe précédent. Il s'agit ici de présenter les mécanismes de surcharge sous Scilab.

Le terme « surcharge » n'est peut-être pas totalement adapté dans la mesure où il n'est possible, pour des raisons d'efficacité, que d'étendre la définition d'un opérateur ou d'une fonction pour des types de données pour laquelle l'opérateur, ou la fonction, n'est pas défini. Plus simplement, il est possible de donner un sens à l'opération `"toto"+1`, mais il n'est pas possible, sans modifier le code de Scilab, de donner un sens nouveau à `1+1`.

6.2.1 Mécanisme de surcharge des opérateurs

La surcharge des opérateurs unaires (`-` , `'` , `.'`) ou binaires (`-` , `+` , `*` etc.) est relativement simple. L'interpréteur cherche à exécuter une fonction dont le nom est formé comme suit :

```
%<type_du_premier_operande>_<op_code>_<type_du_second_operande>
```

où `<op_code>` est une séquence de caractères associée à l'opérateur selon la table 6.1 et les types d'opérandes sont des séquences de caractères associées aux types des variables comme décrit dans la table 6.2.
Ainsi il est possible de donner du sens à la syntaxe `"toto"+1` en définissant la fonction `%s_a_c` :

```
->function r=%c_a_s(a,b),r=a+string(b),endfunction
->"toto"+1
 ans  =

 toto1
```

'	t	./.	y	<>	n
+	a	.	z	|	g
-	s	:	b	&	h
*	m	.*	u	.^	j
/	r	/.	v	~	5
\	l	\.	w	.'	0
^	p	[a,b]	c	<	1
.*	x	[a ;b]	f	>	2
./	d	()extraction	e	<=	3
.\	q	()insertion	i	>=	4
.*.	k	==	o		

Table 6.1 – Surcharge : nom symbolique des opérateurs.

Matrice de chaînes de caractères	c
Matrice de polynômes	p
Matrice de nombres flottants	s
Matrice de booléens	b
Matrice creuse de nombres flottants	sp
Matrice creuse de booléens	spb
Matrice d'entiers	i
Matrice de `handle`	h
Structure `list`	l
Structure `tlist`	<tlist_type>
Structure `mlist`	<mlist_type>
Fonction	m
Fonction compilée	mc

Table 6.2 – Surcharge : nom symbolique des opérandes.

Pour les opérations n-aires que sont l'insertion et l'extraction, le même mécanisme fonctionne mais sans prendre en compte le type des indices. Le nom de la fonction associé à l'extraction est :

```
%<type_de_l'objet>_e
```

et celui associé à l'insertion est :

```
%<type_de_l'objet_source>_i_<type_de_l'objet_cible>
```

Ainsi la fonction `%s_i_c` permet de donner du sens à la syntaxe `A(2)=1` où `A` est une matrice de chaînes de caractères :

```
->function cible=%s_i_c(i,source,cible)
->   cible(i)=string(source)
```

```
->endfunction

->A="texte";

->A(2)=1
 A  =

!texte  !
!       !
!1      !
```

6.2.2 Mécanisme de surcharge des fonctions

Pour les fonctions, le mécanisme de surcharge est un peu moins naturel. La difficulté pour les fonctions réside dans le nombre d'arguments d'entrée qui peut être assez grand ; il est difficilement envisageable de construire des noms de fonctions de surcharge formés de l'ensemble des types des arguments. Pour les fonctions à un seul argument le nom de la fonction de surcharge se construit tout simplement comme :

```
%<type_de_l'objet>_<nom_de_la_fonction>
```

Ainsi la fonction `%c_sin` permet de donner du sens à la syntaxe `sin("toto")` :

```
->function r=%c_sin(a),r="sin("+a+")",endfunction
->sin("toto")
 ans  =

 sin(toto)
```

Dans les cas plus compliqués, il suffit de laisser le soin à Scilab de vous fournir le nom :

```
->plot2d(1:10,(%s+1)/(%s-1))
                           !-error   246
function not defined for given argument type(s),
  check arguments or define function %r_plot2d for overloading
->function %r_plot2d(x,y,varargin)
->  plot2d(x,horner(y,x),varargin(:))
->endfunction
->plot2d(0:0.03:5,(%s+2)/(%s^4+2))
```

et l'on obtient la figure 6.1. On notera l'usage de la fonction `horner` qui permet d'évaluer un polynôme ou une fraction rationnelle en tous les points du vecteur fourni par le deuxième argument.

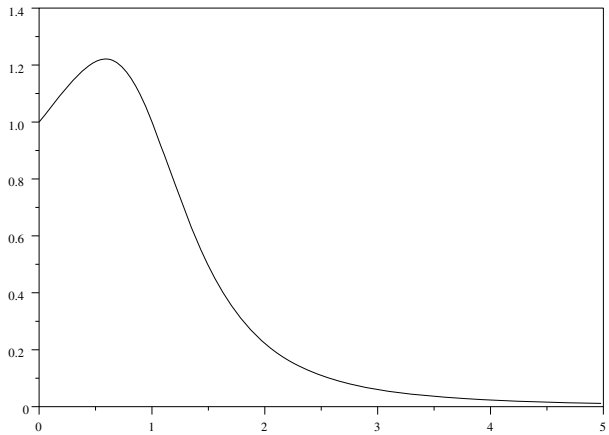


Figure 6.1 – Graphe d'une fonction rationnelle.

6.3 Bibliothèques de fonctions

Lorsqu'un utilisateur a défini plusieurs fonctions, il souhaite en général pouvoir les charger dans Scilab de la façon la plus transparente possible et leur associer une aide en ligne.

6.3.1 Définir une bibliothèque de fonctions

Il est bien entendu toujours possible d'ajouter l'ensemble des instructions `exec` (voir le paragraphe 3.3) dans le fichier de startup (voir le paragraphe 3.1) ou d'utiliser la fonction `getd` qui charge tous les fichiers d'extension `.sci` contenus dans un répertoire, mais cela présente l'inconvénient d'encombrer l'espace des variables de Scilab avec l'ensemble des fonctions définies.

Les bibliothèques de fonctions permettent d'éviter cet inconvénient en ne définissant dans l'espace des variables qu'une seule variable de type `library` qui contient le chemin d'un répertoire et les noms des fonctions de ce répertoire. Pour créer une telle bibliothèque, il faut avoir au préalable généré une sauvegarde binaire de chacune des fonctions de la bibliothèque. Cela pourrait se faire manuellement par une succession d'instructions `exec` et `save`. Pour générer une bibliothèque il suffit alors de créer dans le répertoire un fichier de nom `names` contenant la liste des fonctions et d'utiliser la fonction Scilab `lib` pour créer la variable de type `library`. Heureusement toute cette procédure est automatisée par la fonction `genlib`.

La fonction `genlib` a deux arguments, le premier est une chaîne de caractères qui donne le nom de la variable de type `library` à créer ou à mettre à jour, le second est une chaîne de caractères donnant le chemin du répertoire contenant les fichiers de définition des fonctions. Pour chaque fichier d'extension `.sci`, la fonction `genlib` génère ou met à jour le fichier `.bin` correspondant contenant la sauvegarde binaire de la ou des fonctions définies, puis génère le fichier `names` et enfin appelle la fonction `lib` pour charger la bibliothèque. Une fois la bibliothèque définie, toute référence à l'une de ses fonctions en provoquera le chargement automatique. Une fois chargée, la fonction demeure dans l'espace des variables et suit la règle commune aux autres variables. Une fois les fichiers `.bin` créés, les fichiers `.sci` ne sont plus nécessaires, sauf pour réaliser des modifications ultérieures. Il est possible de mettre à jour une bibliothèque (`library`) en appelant la fonction `genlib` avec comme seul argument son nom. Ainsi `genlib('elemlib')` ou plus simplement encore `genlib elemlib` met à jour la bibliothèque `elemlib`.

La définition de bibliothèques nécessite de respecter quelques règles simples :
- le nom du fichier d'extension `.bin` doit être le nom de la fonction principale qu'il définit ;
- dans le cas où ce fichier contient la définition de plusieurs fonctions, les fonctions secondaires ne sont chargées que lors du référencement de la fonction principale ;
- si deux bibliothèques définissent la même fonction principale, le référencement de cette fonction provoque le chargement de celle définie dans la bibliothèque la plus récemment chargée.

Donnons un exemple d'école pour illustrer le fonctionnement des bibliothèques.

Supposons que le répertoire `testbib` contienne le fichier `fonc1.sci` :

```
function y=fonc1(a,b)
  y=a+sin(b)
endfunction
```

Fichier source : `scilab/testbib/fonc1.sci`

et le fichier `fonc2.sci` :

```
function y=fonc2(a,b)
  y=a+cos(b)
endfunction
```

Fichier source : `scilab/testbib/fonc2.sci`

Alors :

```
->//création et chargement de malib
->genlib("malib","scilab/testbib")
```

```
->malib //visualisation
 malib  =

 Functions files location :scilab/testbib/

 fonc1     fonc2

->fonc1(2,%pi/2) //la fonction est chargée et exécutée
 ans  =

    3.

->fonc1(1,%pi) //la fonction est exécutée
 ans  =

    1.
->clear fonc1 // fonc1 est éliminée de l'espace des variables
->disp(fonc1) //la fonction est re-chargée
 fonc1  =

[y]=fonc1(a,b)

->fonc2(2,%pi/4) //fonc1 est chargée et exécutée
 ans  =

    2.7071068
```

L'instruction `genlib("malib","scilab/testbib")` crée la variable `malib` mais ne charge pas les fonctions elles-mêmes. La fonction `string` appliquée à une variable de type `library` retourne un vecteur de chaînes contenant le chemin du répertoire puis les noms des fonctions principales.

```
->txt=string(malib)
 txt  =

!scilab/testbib/  !
!                 !
!fonc1            !
!                 !
!fonc2            !
```

6.3.2 Définition de l'aide en ligne

Pour être réellement utilisable, une fonction doit être documentée par une aide en ligne. Nous expliquons ici comment réaliser cette documentation.

La façon la plus simple de documenter une fonction scilab est d'insérer un ensemble de commentaires immédiatement après la ligne de définition de la syntaxe de la fonction. Ces commentaires (jusqu'à la première ligne ne débutant

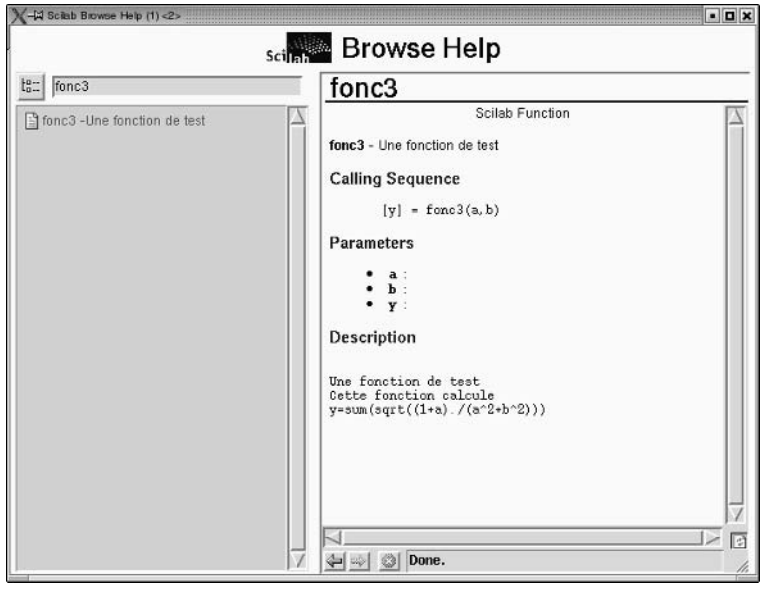


Figure 6.2 – Exemple de rendu d'aide minimal.

pas par \\ ) seront alors automatiquement extraits et affichés dans l'outil de visualisation de l'aide en ligne.

Supposons que l'on ait défini la fonction `fonc3` :

```
function y=fonc3(a,b)
//Une fonction de test
//Cette fonction calcule  y=sum(sqrt((1+a)./(a^2+b^2)))
  y=sum(sqrt((1+a)./(a^2+b^2)))
endfunction
```

Fichier source : `scilab/fonc3.sci`

La commande `help fonc3` affichera la fenêtre et l'on obtient la figure 6.2. Pour obtenir une aide hypertexte, mieux formatée et pouvant contenir des images, les concepteurs de Scilab ont choisi de documenter les fonctions dans des fichiers différents des fichiers de définition des fonctions. Ces fichiers sont alors écrits en HTML ou mieux en XML.

Pour rédiger la documentation d'une fonction sans trop avoir à se préoccuper du langage XML, il existe dans Scilab une fonction qui permet de générer le squelette XML associé à la documentation de la fonction. Par exemple :

```
->help_skeleton('fonc3','./')
 ans  =
```

```
./fonc3.xml
```

produit le fichier XML suivant :

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<!DOCTYPE MAN SYSTEM "file:///data/scilab-4.0/man/manrev.dtd">
<MAN>
  <LANGUAGE>eng</LANGUAGE>
  <TITLE>fonc3</TITLE>
  <TYPE>Scilab Function  </TYPE>
  <DATE>7-Jul-2006</DATE>
  <SHORT_DESCRIPTION name="fonc3">
     add short decription here
  </SHORT_DESCRIPTION>

  <CALLING_SEQUENCE>
  <CALLING_SEQUENCE_ITEM>y = fonc3(a,b)</CALLING_SEQUENCE_ITEM>
  </CALLING_SEQUENCE>
  <PARAM>
  <PARAM_INDENT>
    <PARAM_ITEM>
    <PARAM_NAME>a</PARAM_NAME>
    <PARAM_DESCRIPTION>
       <SP>
       : add here the parameter description
       </SP>
    </PARAM_DESCRIPTION>
    </PARAM_ITEM>
    <PARAM_ITEM>
    <PARAM_NAME>b</PARAM_NAME>
    <PARAM_DESCRIPTION>
       <SP>
       : add here the parameter description
       </SP>
    </PARAM_DESCRIPTION>
    </PARAM_ITEM>
    <PARAM_ITEM>
    <PARAM_NAME>y</PARAM_NAME>
    <PARAM_DESCRIPTION>
       <SP>
       : add here the parameter description
       </SP>
    </PARAM_DESCRIPTION>
    </PARAM_ITEM>
  </PARAM_INDENT>
  </PARAM>

  <DESCRIPTION>
     <DESCRIPTION_INDENT>
```

```
    <DESCRIPTION_ITEM>
    <P>
     Add here a paragraph of the function description.
     Other paragraph can be added
    </P>
    </DESCRIPTION_ITEM>
    <DESCRIPTION_ITEM>
    <P>
     Add here a paragraph of the function description
    </P>
    </DESCRIPTION_ITEM>
    </DESCRIPTION_INDENT>
  </DESCRIPTION>

  <EXAMPLE><![CDATA[
   Add here scilab instructions and comments
  ]]></EXAMPLE>

  <SEE_ALSO>
    <SEE_ALSO_ITEM> <LINK> add a key here</LINK> </SEE_ALSO_ITEM>
    <SEE_ALSO_ITEM> <LINK> add a key here</LINK> </SEE_ALSO_ITEM>
  </SEE_ALSO>

  <AUTHORS>
    <AUTHORS_ITEM label='enter here the author name'>
    Add here the author  references
    </AUTHORS_ITEM>
  </AUTHORS>

  <BIBLIO>
    <SP>
    Add here the function bibliography if any
    </SP>
  </BIBLIO>

  <USED_FUNCTIONS>
     <SP>
     Add here the used function name and  references
     </SP>
  </USED_FUNCTIONS>
</MAN>
```

Fichier source : `help/fonc3_sql.xml`

L'utilisateur peut alors, avec son éditeur favori ou avec un éditeur XML comme xxe[1], remplacer les lignes `Add here...` par les informations relatives à la fonction. La fonction `xmltohtml` avec comme argument le chemin du ré-

[1] www.xmlmind.com/xmleditor.

pertoire contenant les fichiers XML convertira ces fichiers en HTML[2]. Il suffit alors d'utiliser la fonction `add_help_chapter` pour ajouter le répertoire ci-dessus comme nouveau chapitre du système d'aide en ligne.

On peut ainsi obtenir simplement le rendu de la figure 6.3 dans la fenêtre d'aide. L'équation $\sum_i \sqrt{\frac{1+a_i}{a_i^2+b_i^2}}$ est insérée comme une image produite par l'utilitaire `latex2gif`. On remarquera les liens hypertexte vers la documentation des fonctions `sum` et `sqrt`.

6.4 Erreurs et gestion des erreurs

6.4.1 Généralités

Dans les logiciels interprétés comme Scilab, de nombreuses vérifications de cohérence des instructions sont effectuées au cours de l'exécution. Les premières vérifications effectuées concernent la vérification syntaxique, puis l'existence des variables référencées et enfin, pour chacune des primitives, les programmes d'interface des primitives (voir le paragraphe 7) vérifient la cohérence du nombre d'arguments d'entrée, de sortie, la cohérence du type et des dimensions des arguments d'entrées, l'espace mémoire disponible, etc.

La détection de l'une de ces erreurs déclenche alors un mécanisme de gestion de l'erreur. Par défaut ce mécanisme effectue les étapes suivantes :

1. édition du message d'erreur correspondant ;
2. édition de la séquence d'appel des fonctions ;
3. arrêt de l'évaluation en cours ;
4. réinitialisation de l'interprète.

```
->function y=foo(a),y=spec(a)+1,endfunction

->foo(rand(2,3))
 !-error    20
first argument must be square matrix at line       2 of function foo
called by : foo(rand(2,3))
 ->
```

Si l'instruction qui a produit l'erreur a été saisie après une `pause` (voir la section 3.5), alors le gestionnaire d'erreur rend la main au même niveau :

```
-1->foo(rand(2,3))
 !-error    20
first argument must be square matrix at line       2 of function foo
called by : foo(rand(2,3))
-1->
```

[2]Cette fonction s'appuie sur l'utilitaire Sablotron (www.gingerall.com) et Xsltproc (xmlsoft.org).

fonc3

Scilab Function

Last update : 11/7/2006

fonc3 - Une fonction test

Calling Sequence

```
y = fonc3(a,b)
```

Parameters

- **a** : a real vector
- **b** : a real vector with same size as **a**
- **y** : a real scalar

Description

La fonction **fonc3** permet de calculer l'expression: $\sum_i \sqrt{\frac{1+a_i}{a_i^2+b_i^2}}$

Examples

```
a = 1:3; b = a;
y=fonc3(a,b)
```

See Also

sqrt , **sum** ,

Authors

Dr Scilab www.scilab.org

Used Function

sqrt, sum

Figure 6.3 – Exemple de rendu d'aide XML.

6.4.2 Instruction try-catch

La structure de contrôle `try-catch` (voir page 37 permet de rattraper des erreurs d'exécution. D'autres utilitaires de gestion d'erreur sont disponibles pour l'aide à la mise au point ou pour des utilisations plus avancées.

6.4.3 La fonction errcatch

La fonction `errcatch` permet de modifier le mécanisme de gestion des erreurs. Le mode de gestion par défaut correspond à l'option `kill` mais il existe trois autres options :

- l'option `pause` qui au lieu d'effectuer les étapes 3 et 4 ci-dessus, saute l'instruction en cours (ici `y=spec(a)+1`) et provoque une `pause` :

```
->function y=foo(a),y=spec(a)+1,endfunction

->errcatch(20,"pause")

->foo(rand(2,3))
 !-error    20
first argument must be square matrix


-1->a
 a  =

!   0.2113249    0.0002211    0.6653811 !
!   0.7560439    0.3303271    0.6283918 !
-1->
```

  Le premier argument de `errcatch` est le numéro de l'erreur que l'on souhaite contrôler. Si cet argument vaut `-1` alors toutes les erreurs sont contrôlées.
  Il est alors possible de terminer l'exécution par l'instruction `abort`, ou de la poursuivre en ayant ou non modifié des variables :

```
-1-> a=a(:,1:2) ;
-1-> y=resume(spec(a)+1)
 ans  =

!   1.2099362 !
!   1.3317158 !

->
```

  Cette option peut être très utile pour le débogage (voir aussi le paragraphe 3.5), dans la mesure où elle rend la main à l'utilisateur dans le contexte de l'erreur.
- l'option `continue`, au lieu d'effectuer les étapes 3 et 4 ci-dessus, saute l'instruction en cours (ici `y=spec(a)+1`) et reprend l'exécution :

```
->function y=foo(a)
->   errcatch(20,"continue","nomessage")
->   y=spec(a)+1;
->   errcatch()
->   if iserror() then errclear(),y=[],end
->endfunction

->foo(rand(2,3))
 ans  =

     []
```

L'option supplémentaire `"nomessage"` indique que l'affichage du message d'erreur doit être inhibé. L'instruction `errcatch()` revient au mode de gestion par défaut, l'expression `iserror()` retourne 1 si une erreur s'est produite tandis que `errclear` réinitialise la mémoire de `iserror`.
Le message associé à la dernière erreur détectée peut être récupéré ultérieurement par la fonction `lasterror`.

```
->lasterror()
 ans  =

 first argument must be square matrix
```

– l'option `stop` est beaucoup plus radicale : elle provoque l'arrêt de Scilab.

6.4.4 La fonction execstr

La gestion des erreurs peut aussi s'effectuer avec la fonction `execstr` que l'on a déjà utilisée (voir page 43). La fonction `foo` précédente pourrait s'écrire :

```
->function y=foo(a)
->  ierr=execstr("y=spec(a)+1","errcatch");
->  if ierr~=0 then y=[],end
->endfunction

->foo(rand(3,2))
 ans  =

     []
->lasterror()
 ans  =

 first argument must be square matrix
```

La fonction `lasterror` permet de retrouver le dernier message d'erreur.

Certaines fonctions comme `mopen` peuvent retourner une variable contenant un indicateur d'erreur au lieu de déclencher le mécanisme de gestion des erreurs. Dans ce cas le programmeur doit gérer lui même les conséquences de l'erreur.

Chapitre 7

Interfaçage

7.1 Écriture d'une interface

Supposons que l'on dispose d'une bibliothèque de fonctions externes écrites en C. La bibliothèque contient par exemple la fonction `geom` qui effectue `m*n` tirages aléatoires selon une loi géométrique de paramètre `p` et stocke les résultats dans un tableau d'entiers `y` de taille `m` x `n`. Savoir ce que fait exactement `geom` n'est pas vraiment utile pour nous ici. Disons simplement que, étant donnés les entiers `m` et `n` et le paramètre `p`, cette fonction retourne le tableau `y`. Voilà le code C de la procédure `geom` (qui est fort simple), la seule chose importante ici étant sa liste d'appel :

```
#include <stdlib.h>

int geom(int m,int n,double p,int y[])
{
  int i;
  if ( p >= 1.0 )
    {
      cerro("p doit-etre < 1 "); return 1;
    }
  for (i = 0 ; i < m*n ; i++)
    {
      double z = drand48();
      y[i] = 1;
      while ( z < 1.0-p ) { z = drand48(); y[i] ++; }
    }
  return 0;
}
```

Fichier source : `C/geom.c`

On souhaite ajouter dans Scilab une fonction (une primitive) de même nom `geom` dont la syntaxe d'appel est `y=geom(m,n,p)`. La fonction `geom` est censée

effectuer `m*n` tirages de loi géométrique et renvoyer le résultat sous la forme d'une matrice Scilab de taille `m` x `n`.

Pour ce faire, il faut écrire une interface, c'est-à-dire un programme qui va être chargé de faire les conversions et le passage des données entre l'interprète Scilab et la procédure C `geom`. Nous appellerons `intgeom` l'interface pour la fonction `geom` (il faut écrire une interface pour chaque fonction que l'on veut interfacer).

Il existe dans Scilab diverses façons d'écrire une interface. Nous allons utiliser ici la méthode qui nous semble la plus simple. De nombreux exemples d'écriture d'interfaces, destinés aux utilisateurs débutants, sont fournis dans le répertoire : `SCI/examples/interface-tutorial-so/`.

Pour écrire cette interface particulière, on a utilisé des procédures de la bibliothèque d'interfaçage (`CheckRhs`, `CheckLhs`, `GetRhsVar` et `CreateVar`), des variables de la bibliothèque d'interfaçage (`LhsVar`) et des fonctions internes Scilab (`Scierror`, aussi utilisée dans `geom`) (`Rhs` et `Lhs` désignent respectivement le membre droit et le membre gauche d'une égalité ou d'une équation). Pour utiliser les fonctions et les variables précédentes, il faut rajouter le fichier d'entête `stack-c.h` dans le fichier `intgeom.c`.

L'interface de la fonction `geom` s'écrit alors de la façon suivante :

```
#include "stack-c.h"

extern int geom(int m, int n, double p, int y[]);

int intgeom(char *fname)
{
  int l1, m1, n1, l2, m2, n2, m3, n3, l3;
  int minlhs=1, maxlhs=1, minrhs=3, maxrhs=3;
  int m,n,y;  double p;

  /* 1 - Vérification du nombre d'arguments  */
  CheckRhs(minrhs,maxrhs) ; /* d'entrée      */
  CheckLhs(minlhs,maxlhs) ; /*  de sortie    */

  /* 2 - Vérification du type des arguments
         et retour des pointeurs             */
  GetRhsVar(1, "i", &m1, &n1, &l1);
  GetRhsVar(2, "i", &m2, &n2, &l2);
  GetRhsVar(3, "d", &m3, &n3, &l3);

  if ( m1*n1 != 1 || m2*n2 != 1 || m3*n3 != 1)
    {
      Scierror(999,
      "Les arguments de geom doivent être des scalaires");
      return 0;
    }
  /* 3 - Extraction des valeurs */
  m = *istk(l1); n = *istk(l2) ; p = *stk(l3);
```

```
  /* 4 - Création de la variable de retour */
  CreateVar(4,"i",&m,&n,&y);
  if ( geom(m,n,p,istk(y)) == 1 ) return 0;

  /* 5 - Spécification de la variable de retour */
  LhsVar(1) = 4;
  return 0;
}
```

Fichier source : `C/intgeom.c`

À première vue, cette fonction peut paraître un peu compliquée. En fait, on ne va pas écrire tout ce code, mais plutôt partir d'un exemple tout fait que l'on va adapter à notre cas. L'interface précédente pourrait par exemple être utilisée presque telle quelle pour n'importe quelle fonction C qui aurait une liste d'appel semblable à celle de `geom`.

Comment cette interface fonctionne-t-elle ? Quand sous l'interprète Scilab, on tape la commande `y=geom(m,n,p)`, les arguments d'entrée sont évalués et leurs valeurs sont stockées dans un tableau (que l'on appellera pile d'appel) dans l'ordre où ils apparaissent dans la liste d'appel. D'autre part le nombre d'arguments de retour attendus (ici `y`) est connu. La fonction d'interfaçage `intgeom` doit tout d'abord contrôler que les nombres d'arguments transmis et attendus au retour sont corrects. Cela est fait en utilisant `CheckLhs` et `CheckRhs` ; si le nombre d'arguments ne correspond pas aux bornes spécifiées, ces fonctions génèrent une erreur et provoquent la sortie de l'interface. Dans cet exemple, on attend exactement 3 variables d'entrée (`minrhs=3` et `maxrhs=3`) et une variable de sortie (`minlhs=1` et `maxlhs=1`).

Dans l'interface, chaque variable Scilab est repérée par un numéro, d'abord les variables d'entrée puis les variables de sortie. Ici par exemple, la fonction Scilab étant `y=geom(m,n,p)`, les variables d'entrée `m`, `n`, `p` ont les numéros 1, 2, 3 et la variable de sortie `y` va avoir le numéro 4.

Après avoir vérifié que la fonction est appelée avec le bon nombre d'arguments, il faut vérifier que chaque argument présent dans la liste d'appel a le bon type (est-ce une matrice, une chaîne de caractères, une fonction ?) et la bonne taille. Enfin, il faut récupérer un pointeur vers les données de sortie de `geom` pour pouvoir les transmettre à la fonction interfacée ; c'est ce qui est fait dans la deuxième partie de la procédure `intgeom`.

La commande suivante :

```
  GetRhsVar(2, "i", &m2, &n2, &l2);
```

a pour effet de vérifier que le deuxième argument de la liste d'appel (`n`) est bien de type numérique entier (`"i"`) et de renvoyer dans `m2` et `n2` les dimensions du tableau passé en deuxième argument (`n`). L'entier `l2` est une adresse pour

accéder aux données (une conversion des données en entiers est faite). Si le type ne convient pas une erreur est générée. Les deux premiers arguments de `GetRhsVar`, numéro de la variable d'entrée et type, sont des arguments d'entrée de `GetRhsVar` alors que les trois derniers arguments sont des arguments de sortie de `GetRhsVar`.

Le deuxième argument étant de type entier `istk(l2)` est le pointeur sur cet entier. Le deuxième argument de `geom` qui doit être un entier est donc finalement récupéré au niveau C dans l'interface par `n= *istk(l2)`.

On notera qu'il est prudent de vérifier que l'argument 2 fourni à l'appel de la fonction Scilab `geom` est bien scalaire (tableau de dimension 1×1). Cela peut être fait en vérifiant que `m2*n2` vaut bien 1.

Le troisième argument est de type double (`"d"`) et on y accède par `p=*stk(l3)`, `p` étant déclaré `double` et `stk(l3)` étant un pointeur de doubles. On peut noter ici que l'on récupérerait une chaîne de caractères selon le même principe par l'appel

```
GetRhsVar{k,"c", &m, &n, &l)
```

et `cstk(l)` serait alors un pointeur de `char` sur la chaîne passée. De manière générale, disposant des dimensions des arguments, on doit effectuer des vérifications et en cas d'erreur on peut utiliser la fonction `Scierror` pour afficher un message puis faire un `return` (Scilab prend alors le relais). On notera d'ailleurs que, dans la fonction `geom`, on a aussi utilisé la fonction `Scierror` pour imprimer un message d'erreur.

Avant d'appeler la fonction `geom`, il faut créer la variable numéro 4, en l'occurrence `y`, et réserver de la place pour la stocker. La variable `y` doit être ici une matrice d'entiers de taille `m`×`n` et la fonction `geom` veut un pointeur d'entiers. La commande :

```
CreateVar(4,"i",&m,&n,&y)
```

de la quatrième partie de `intgeom` se charge de créer dans la pile d'appel un nouvel objet Scilab avec le numéro `4` (une matrice `m`×`n`) et à nouveau on obtient une adresse pour accéder aux données `y` (les données sont entières et sont donc accessibles via `istk(y)`).

La syntaxe d'appel de `CreateVar` est la même que celle de `GetRhsVar` sauf que les 4 premiers arguments sont des entrées et le dernier est une sortie calculée par `CreateVar`. La fonction `CreateVar` crée une variable Scilab de type matrice qui doit être ensuite remplie (colonne par colonne). Ici `geom` donne des valeurs entières à `istk(y)[0]`, `istk(y)[1]`, etc. Après l'appel de `geom`, il ne reste plus qu'à renvoyer à Scilab le résultat, ce que réalise la cinquième partie de la procédure `intgeom`. Le nombre de variables attendues par Scilab a déjà été contrôlé au début de l'interface (`CheckRhs`) et il ne reste plus qu'à indiquer par l'intermédiaire de `LhsVar(i)=j` les positions des variables que l'on veut renvoyer : l'instruction `LhsVar(1)=4` signifie « la première variable de sortie est la variable

numéro 4 ». C'est Scilab qui contrôle alors tout seul d'éventuelles conversions de données à effectuer.

Il n'est bien sûr pas question de décrire ici toutes les fonctions de la bibliothèque d'interfaçage. Le lecteur intéressé pourra se reporter au répertoire
`SCI/examples/interface-tour-so`
pour une description au travers d'exemples de toutes les possibilités de la bibliothèque. La connaissance des 4 fonctions (`CheckLhs`, `CheckRhs`, `GetRhsVar`, `CreateVar`) permet d'interfacer la plupart des fonctions C.

Dans le répertoire `SCI/examples/interface-tour-so`, on montre en particulier comment interfacer une fonction C qui a elle-même une fonction comme paramètre d'entrée, ou comment on peut appeler l'interprète Scilab à l'intérieur d'une interface. C'est évidemment un peu plus compliqué, mais des exemples simples sont donnés pour réaliser ce type d'interface.

Les utilisateurs familiers des `mexfiles` Matlab pourront se reporter au répertoire `SCI/examples/mex-examples` pour constater que leurs fichiers `mex` (bibliothèque d'interfaçage de Matlab) marchent quasiment tels quels dans Scilab.

7.2 Chargement et utilisation

Il nous reste une deuxième étape importante : comment charger et utiliser le code précédent dans Scilab. Nous allons suivre ici la méthode qui est utilisée dans le répertoire `SCI/examples/interface-tour-so`. Pour que l'exemple soit plus réaliste, nous allons interfacer deux fonctions : la fonction `geom`, que nous avons vue précédemment, et la fonction système `srand48`, qui permet d'initialiser le générateur aléatoire `drand48`. La fonction système `srand48` ne renvoie pas de valeur et admet pour argument un entier, voilà son interface :

```
int intsrand48(char *fname)
{
  int l1, m1, n1;
  CheckRhs(1,1);
  CheckLhs(0,1);
  GetRhsVar(1, "d", &m1, &n1, &l1);
  if ( m1*n1 != 1 ) {
      cerro("Erreur: srand48 attend un argument scalaire");
      return 0;
  }
  srand48((long int) *stk(l1));
  LhsVar(1) = 0; /* pas de valeur renvoyée */
  return 0;
}
```

Fichier source : `C/srand48.c`

Nous avons donc deux fonctions à interfacer ; nous supposerons que les codes C correspondants sont écrits dans un unique fichier `intgeom.c`, contenant

intgeom, geom et intsrand48. On notera qu'il n'y a en fait aucune contrainte sur la répartition des fonctions dans les fichiers. Il faut maintenant charger ces codes dans Scilab pour définir deux nouvelles primitives geom et srand48. Pour cela, il faut compiler le code, le charger dynamiquement (en chargeant une bibliothèque partagée .so sous Unix ou .dll sous Windows) et indiquer les noms Scilab (geom et srand48) que l'on veut donner aux fonctions dont les codes sont interfacés par intgeom et intsrand48.

Tout cela va pouvoir s'effectuer depuis Scilab d'une façon indépendante du système hôte (Unix, Windows) et des compilateurs disponibles de la façon suivante. Au moyen d'un éditeur de texte, on va constituer un fichier de nom builder.sce. Ce nom est canonique et un utilisateur qui rencontre un fichier de nom builder.sce sait qu'il faut exécuter un tel fichier pour configurer une contribution, une interface, etc.

```
mode(-1) //suppression de la visualisation
// Le chemin absolu du fichier
path = get_absolute_file_path("builder.sce")
old=pwd();cd(path) //
//Le nom de la librarie dynamique
ilib_name  = "libalea"
//Le vecteur des noms des fichiers objets
files = ["intgeom.o" "intsrand48.o" "geom.o"];
libs  = [] //autres libraries eventuelles

// table des couples (<nom scilab> <nom d'interface>)

table =["geom", "intgeom";
        "srand48","intsrand48"];

ilib_build(ilib_name,table,files,libs,"Makefile")
cd(old);
clear old ilib_name path files libs table
```

Fichier source : C/builder.sce

Une base de départ pour construire le fichier builder.sce est le fichier d'exemple que l'on trouvera dans le répertoire :

SCI/examples/interface-tutorial-so

que l'on peut facilement adapter à un nouvel interfaçage.

Dans ce fichier on trouve les informations suivantes :

- ilib_name contient une chaîne de caractères, c'est le nom (lib...) que l'on veut donner à la bibliothèque (un ensemble d'interfaces) que l'on est en train de construire.
- files est un vecteur de chaînes de caractères donnant la liste des fichiers objets qui constituent la bibliothèque. Ici on trouve juste le fichier intgeom.o. Les fichiers objets sont notés avec un suffixe .o et cela même si l'on se trouve sous Windows. Un même fichier builder.sce doit pouvoir s'exécuter sans modifications sous Windows ou sous Unix.

- `libs` est une chaîne de caractères donnant une liste de bibliothèques nécessaires à la création de la bibliothèque partagée (sans extensions).
- `table` est une matrice de chaînes de caractères où l'on donne la correspondance entre un nom de fonction Scilab et un nom d'interface. On trouve par exemple sur la première ligne `"geom", "intgeom";` qui indique que la fonction Scilab `geom` est interfacée au moyen de l'interface `intgeom`

Il suffit alors de faire exécuter ce fichier `builder.sce` par Scilab pour

- créer une bibliothèque partagée (en général `.so` sous Unix/Linux et `.dll` sous Windows) ;
- créer un fichier `loader.sce` qui permettra de charger la bibliothèque dans Scilab.

L'exécution de ce fichier (ici sous Linux) donne :

```
->exec C/builder.sce;
   generate a gateway file
   generate a loader file
   generate a Makefile: Makelib
   running the makefile
   compilation of intgeom
   compilation of intsrand48
   compilation of geom
   building shared library (be patient)
```

Après l'exécution de ce fichier, plusieurs nouveaux fichiers sont créés dans le répertoire courant.

Le plus important d'entre eux est le fichier `loader.sce`. Il permet le chargement dynamique dans Scilab de la nouvelle bibliothèque au moyen de la primitive `addinter`. En pratique, on exécute donc une fois `builder.sce` pour compiler la bibliothèque puis à chaque nouvelle session Scilab on exécute seulement `loader.sce` pour charger la bibliothèque (noter que le chargement peut être effectué de n'importe quel répertoire, il n'est pas nécessaire de se trouver dans le répertoire de `loader.sce` pour l'exécuter). Évidemment, on peut aussi créer un fichier de démarrage `.scilab` ou `scilab.ini` qui contient la commande `exec loader.sce`. Voici le contenu du fichier `loader.sce` créé par l'exécution de `builder.sce` pour notre exemple (dans un environnement Linux).

```
 libalea_path=get_file_path("loader.sce");
functions=[ "geom";
            "srand48"];
addinter(libalea_path+"/libalea.so","libalea",functions);
```

Il ne reste plus qu'à exécuter ce fichier, ce qui donne ici :

```
->exec C/loader.sce ;
shared archive loaded
```

Citons pour information les autres fichiers générés. On trouve un fichier `libalea.c` appelé « gateway ». Il contient une procédure appelée « procédure

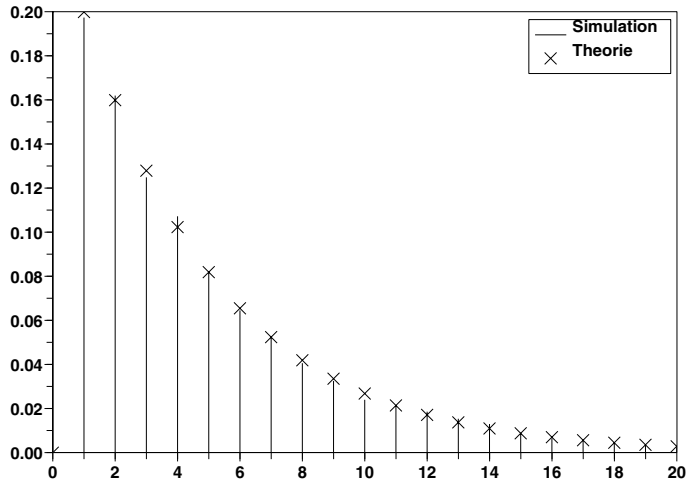


Figure 7.1 – Histogramme des données.

de gestion des interfaces ». Elle sert d'aiguillage pour gérer la table de correspondance `table`.

On trouve également un fichier `Makelib` qui est un fichier `Makefile` qui dépend du système hôte et de l'environnement de compilation (Unix, Windows/Visual C++, Windows/Absoft). Il permet la création de la bibliothèque partagée.

Il ne reste plus qu'à tester cette nouvelle primitive `geom` avec le petit script qui suit. On compare graphiquement sur la figure 7.1 l'histogramme empirique obtenu par simulation et celui donnée par la théorie.

```
n=10000; pr=0.2 ;
y=geom(1,n,pr);   // appel de la nouvelle primitive
N=20;  i=0:N;
//tests des résultats
z=[]; for i1=i, z=[z,size(find(y==i1),"*")];end
plot2d3(i',z'/n);

zt=0;for i1=1:N; zt=[zt,pr*(1-pr)^(i1-1)];end
plot2d1(i',zt',style=-2);
legend(["Simulation","Theorie"])
```

Fichier source : `scilab/geom.sce`

7.3 Utilitaire intersci

Il existe dans Scilab un utilitaire appelé `intersci` qui permet de générer un fichier d'interface à partir d'un fichier de description de la fonction à interfacer. `intersci` ne permet pas toute la souplesse d'une interface écrite à la main en C mais peut cependant être utilisé pour générer des interfaces pour un grand nombre de fonctions. Par exemple, toute la boîte à outils Metanet pour les graphes et les réseaux, qui contient des dizaines de fonctions C et FORTRAN a été liée à Scilab en utilisant `intersci`.

La documentation complète de l'utilisation d'`intersci` se trouve dans le répertoire `doc` de Scilab et de nombreux exemples sont décrits dans le répertoire :

```
SCI/examples/intersci-examples-so
```

Nous nous limitons ici à un exemple simple, interfacer la fonction `ext1c` :

```
#include "machine.h"

int F2C(ext1c)(n, a, b, c)
     int *n;
     double *a, *b, *c;
{
  int k;
  for (k = 0; k < *n; ++k)
      c[k] = a[k] + b[k];
  return(0);
}
```

Fichier source : `C/ext1c.c`

Cette fonction n'a que des arguments de type pointeur et le nom de la fonction est encapsulé dans un appel de macros `C2F(ext1c)` [1].

Pour interfacer cette fonction il faut écrire un fichier de description que nous appelons `ex01fi.desc` (le suffixe `.desc` est imposé) :

```
ext1c a b
a  vector m
b  vector m
c vector  m

ext1c   m a b c
m integer
a double
b double
c double
```

[1] C'est une limitation de l'utilisation d'`intersci` qui doit interfacer à la fois des fonctions C et des fonctions FORTRAN.

```
out sequence c
**********************
```

Fichier source : `C/ex01fi.desc`

Le texte écrit dans le fichier indique que l'on va créer une primitive Scilab de syntaxe `[c]=ext1c(a,b)` où `a`, `b` et `c` sont trois vecteurs de même taille. La première ligne décrit les arguments d'entrée et la dernière les arguments de sortie. La fonction `ext1c`, pour s'exécuter, doit appeler la fonction C ou FORTRAN `C2F(ext1c)` avec quatre arguments : un pointeur d'entier qui contiendra la taille des vecteurs et trois pointeurs de « double » qui permettront d'accéder aux valeurs contenues dans les vecteurs Scilab `a`, `b` et `c`. La place nécessaire pour stocker `c` est réservée dans l'interface avant l'appel de la fonction C `C2F(ext1c)`.

En exécutant `intersci` au moyen de la commande :

```
SCI/bin/intersci-n ex01fi
```

on génère d'abord l'interface `ex01fi.c` :

```
#include "stack-c.h"
/*****************************************
 * SCILAB function : ext1c, fin = 1
 *****************************************/

int intsext1c(fname)
   char *fname;
{
 int m1,n1,l1,mn1,m2,n2,l2,mn2,un=1,mn3,l3;
 CheckRhs(2,2);
 CheckLhs(1,1);
 /*  checking variable a */
 GetRhsVar(1,"d",&m1,&n1,&l1);
 CheckVector(1,m1,n1);
 mn1=m1*n1;
 /*  checking variable b */
 GetRhsVar(2,"d",&m2,&n2,&l2);
 CheckVector(2,m2,n2);
 mn2=m2*n2;
 /* cross variable size checking */
 CheckDimProp(1,2,m1*n1 != m2*n2);
 CreateVar(3,"d",(un=1,&un),(mn3=n1,&mn3),&l3);
 C2F(ext1c)(&mn1,stk(l1),stk(l2),stk(l3));
 LhsVar(1)= 3;
 return 0;
}
```

Fichier source : `C/ex01fi.c`

puis un fichier « builder » (comme au paragraphe 7.2) `ex01fi_builder.sce` :

```
// appel de intersci-n au moyen d'un Makefile
G_make("ex01fi.c","ex01fi.c");
// execution du builder crée par intersci-n
files = ["ex01fi.o" , "ex01c.o"];
libs  = [] ;
exec("ex01fi_builder.sce");
// execution du loader
exec loader.sce
// test de la fonction
a=[1,2,3];b=[4,5,6];
c=ext1c(a,b);
if norm(c-(a+b)) > %eps then pause,end
```

Fichier source : `C/ex01.sce`

On se reportera au répertoire `SCI/examples/intersci-examples-so` où le même exemple est repris de façon plus complète. L'ensemble des opérations à effectuer est récapitulé dans ce fichier `ex01.sce`.

7.4 Utilisation de la commande link

Pour terminer, il convient de parler d'un cas (encore) plus simple où l'on peut charger des fonctions dynamiquement dans Scilab sans avoir à écrire une interface. On utilise pour cela les fonctions Scilab `link` et `call`. Une contrainte est imposée : la liste d'appel de la fonction C à interfacer ne doit contenir que des pointeurs sur des variables de types double ou entier.

Notre fonction `geom` deviendra ici :

```
#include <stdlib.h>
int geom1(int *m, int *n, double *p, int y[])
{
  int i;
  if ( *p >= 1.0 )
    {
      Scierror(999,"p doit-etre < 1 "); return 1;
    }
  for (i = 0 ; i < (*m) * (*n) ; i++)
    {
      double z = drand48();
      y[i] = 1;
      while ( z < 1.0 - *p ) { z = drand48(); y[i] ++; }
    }
  return 0;
}
```

Fichier source : `C/geom1.c`

Comme précédemment il faut créer une bibliothèque partagée et la charger cette fois dans Scilab avec la commande `link`. Il existe donc deux façons de charger une bibliothèque partagée dans Scilab mais qui correspondent à deux usages différents :

- `addinter` rajoute des primitives Scilab et nécessite l'écriture d'interfaces : l'interfaçage de la fonction `geom` (voir 7.2) produit un fichier `loader.sce` qui fait appel à `addinter`.
- `link` ne fait que rajouter des fonctions dans la table des fonctions accessibles par certaines primitives de Scilab comme `ode`, `optim`, `fsolve`...et la primitive `call` qui permet d'accéder depuis Scilab à la fonction C (ou FORTRAN) dynamiquement liée à Scilab par la commande `link`.
  Pour appeler une fonction dynamiquement liée à Scilab à partir d'une primitive de type `call`, `ode`, `optim` ou `fsolve`, il suffit de passer le nom de la fonction liée comme un argument de cette primitive.

La commande `link` permet de lier dynamiquement un ensemble de fichiers objets ou toute une bibliothèque avec Scilab. Montrons le contenu d'un fichier `builder_link.sce` permettant de créer une bibliothèque partagée `libgeom1.so` (ou `libgeom1.dll` sous Windows).

```
mode(-1) //suppression de la visualisation
// Le chemin absolu du fichier
path = get_absolute_file_path("builder_link.sce")
old=pwd();cd(path) //
//Le nom de la librarie dynamique


names=["geom1"];    // les points d'entrée à rajouter
                    // ici un seul.
files = "geom1.o";  // les fichiers objets.
flag = "C";         // une fonction C
 // création des fichiers.
ilib_for_link(names,files,[],flag,..
      'Makefile_link','loader_link.sce');
cd(old); //retour dans le repertoire initial
clear old names path files flag
```

Fichier source : `C/builder_link.sce`

Dans ce qui précède `flag="C"` permet de préciser que la fonction `geom1` est écrite en C. Cette précision est nécessaire car sur certains systèmes le compilateur FORTRAN ajoute un « blanc souligné » (underscore) au début ou à la fin des noms des points d'entrée ce que ne fait pas le C.

L'exécution de ce fichier donne alors :

```
->exec C/builder_link.sce;
   generate a loader file
   generate a Makefile: Makelib
   running the makefile
```

```
   compilation of geom1
   building shared library (be patient)
```

On peut charger cette fonction dans Scilab grace au fichier `loader_link.sce` créé ce qui donne ici :

```
 ->exec C/loader_link.sce; // chargement
de la bibliothèque shared archive loaded Link done
```

On peut alors appeler la fonction C `geom1` depuis Scilab et obtenir sa sortie comme une variable Scilab. On doit envoyer à `geom1` les paramètres d'entrée `m`, `n` et `p` et récupérer la matrice de sortie `y`. C'est la commande `call` qui permet de faire cet appel. La syntaxe est un peu longue car on doit fournir les valeurs des variables d'entrée, mais aussi leur type C et leur position dans la liste d'appel de `geom1`. C'est en effet la commande `call` qui doit faire le travail de conversion que nous avions précédemment codé dans une interface. Cela donne par exemple :

```
->m=3;n=4;p=0.3;

->y=call("geom1",m,1,"i",n,2,"i",p,3,"d","out",[m,n],4,"i");
```

Dans cette instruction, on indique en premier argument le nom de la fonction C appelée (c'est aussi le point d'entrée passé à `link`). Les arguments suivants, jusqu'à la chaîne `out`, donnent l'information relative aux variables d'entrée. Les arguments situés après la chaîne `out` donnent, pour leur part, l'information relative aux variables de sortie.

À chaque variable est associé un triplet de paramètres. On interprète par exemple les trois paramètres `m,1,"i"` comme : passer la valeur `m` comme premier argument de `geom` de type `int`. En sortie, on aura : `y` (premier argument à gauche du signe égal) est une matrice de dimension `[m,n]`, quatrième paramètre de `geom`, de type `int`.

Évidemment, pour simplifier l'appel et inclure les tests de cohérence, on peut encapsuler l'instruction `call(...)` dans une fonction Scilab `y=geom(m,n,p)` et en profiter pour faire des tests (par exemple de dimensions) sur les arguments passés à notre fonction C.

Une autre utilisation de la commande link se fait dans le contexte suivant : certaines primitives Scilab, comme par exemple les fonctions `ode`, `fsolve` ou `optim`, admettent des arguments de type fonction que l'on appelle fonctions externes. Par exemple pour chercher les zéros de la fonction $\cos(x) * x^2 - 1$, on peut construire une fonction Scilab :

```
function y=f(x)
  y=cos(x)*x^2-1
endfunction
```

et utiliser ensuite cette fonction `f` comme argument de la fonction `fsolve` [2] :

```
->y0=10; y=fsolve(y0,f)
 y  =

    11.003833
->f(y)
 ans  =

    3.819D-14
```

Parfois on a envie, pour des raisons d'efficacité, de coder ce genre de fonctions dans un langage compilé (C, C++ ou FORTRANR). Dans ce cas, il ne faut pas vraiment écrire une interface, car la fonction `fsolve` a déjà été prévue pour fonctionner avec des fonctions `f` externes. `fsolve` nous impose par contre une contrainte sur la liste d'appel de la fonction `f` qu'elle peut reconnaître (voir le help de `fsolve`). L'exemple précédent pourrait être codé en C sous la forme :

```
#include <math.h>
void f(int *n,double *x,double *fval,int *iflag)
{
  *fval = cos(*x)*(*x)*(*x) - 1;
}
```

Fichier source : `C/f.c`

On procède comme précédemment pour compiler et charger la fonction C, `f`, dans Scilab :

```
->ilib_for_link("f","f.o",[],"c",'Makefile_f','loader_f.sce'); );
   generate a loader file
   generate a Makefile: Makelib
   running the makefile

->exec loader.sce
```

et on indique à `fsolve` que le problème à résoudre concerne la fonction `f` écrite en C en lui passant en argument la chaîne `"f"` :

```
->y0=10; y=fsolve(y0,"f")
 y  =

    11.003833
```

Cette façon de procéder est valable pour la plupart des primitives Scilab admettant des arguments fonctionnels.

[2] Pour une description de la fonction `fsolve`, voir la section 8.3.

Deuxième partie

Exemples d'applications

Chapitre 8

Résolution d'équations et optimisation

8.1 Matrices

Les matrices sont les objets les plus couramment utilisés dans Scilab. Ce sont des tableaux de nombres qui peuvent coder des transformations linéaires. Par exemple, dans l'espace $\mathbb{R}^3$ on peut définir par une matrice A de taille 3×3 et un vecteur b, la transformation qui donne les coordonnées d'un point image (x_n, y_n, z_n) en fonction des coordonnées (x, y, z) du point de départ :

$$\begin{bmatrix} x_n \\ y_n \\ z_n \end{bmatrix} = A \begin{bmatrix} x \\ y \\ z \end{bmatrix} + b$$

La matrice A représente la partie linéaire de la transformation tandis que le vecteur b est associé à la partie affine.

On va s'intéresser aux transformations qui peuvent être obtenues par composition de trois transformations élémentaires : translation, rotation et homothétie. Ces transformations se programment très facilement en Scilab. La translation consiste simplement à ajouter une constante aux coordonnées :

```
function XYZ=translation(vect,xyz)
  //Chaque colonne de xyz = coordonnées d'un point
  //vect est le vecteur directeur de la translation
  XYZ=(vect(:)*ones(1,size(xyz,2))) + xyz
endfunction
```

Fichier source : `scilab/translation.sci`

La rotation peut se décomposer en rotations élémentaires autour de chacun des axes :

```
function XYZ=rotation(angl,xyz)
  angl=angl/180*%pi;// angles de rotation autour des 3 axes
  c=cos(angl);s=sin(angl)
  // autour de l'axes des abscisses
  Rx=[1  0    0;
      0  c(1) s(1);
      0 -s(1) c(1)]
  // autour de l'axes des ordonnées
  Ry=[c(2) 0 s(2);
      0   1  0;
      -s(2) 0 c(2)]
  // autour de la verticale
  Rz=[c(3) s(3) 0;
      -s(3) c(3) 0;
      0    0    1]
  XYZ=Rx*Ry*Rz*xyz
endfunction
```

Fichier source : `scilab/rotation.sci`

L'homothétie de centre quelconque peut se décomposer en une première translation pour changer l'origine, une mise à l'échelle des coordonnées et enfin une translation pour revenir à l'origine initiale.

```
function XYZ=homothetie(centre,f,xyz)
//centre est le centre de l'homothétie
//f      est le vecteur des facteurs d'homothéties
  XYZ=translation(centre,diag(f)*translation(-centre,xyz))
endfunction
```

Fichier source : `scilab/homothetie.sci`

Le script ci-dessous illustre l'usage de ces transformations et donne la figure 8.1. On dessine un objet représenté par des facettes rectangulaires : on donne les coordonnées `xv`, `yv`, `zv` des sommets des facettes. On associe à l'objet une matrice `XYZ` où chaque colonne donne les coordonnées d'un point de l'objet. On effectue des opérations de translation, homothétie et rotation sur la matrice `XYZ`, c'est-à-dire sur chaque colonne de cette matrice.

```
//Génération de l'objet (un vase)
//Génératrice:
xy=[0,1.2,1.6,2.1,2.2,2,1.6,0.9,0.5,0.3,0.3,0.4,0.6,1,1.4,1.7
 0,0,0.1,0.4,0.8,1.1,1.4,1.7,1.9,2.2,2.4,2.7,3,3.3,3.7,3.9]/2;
function [x,y,z]=vase(alp,Z)
  x=xy(1,Z).*cos(alp)
  y=xy(1,Z).*sin(alp)
  z=xy(2,Z).*ones(alp)
endfunction
//Représentation par facettes quadrangulaires :
```

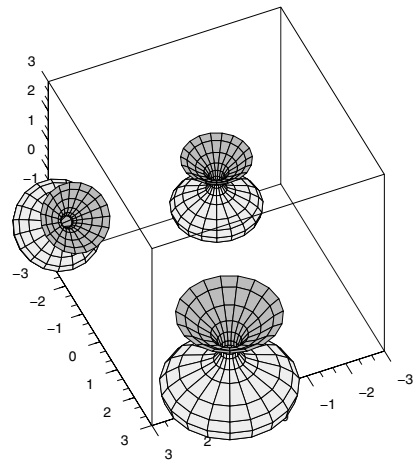


Figure 8.1 – Transformation dans l'espace.

```
// xv = matrice 4 x N des abscisses x des sommets des facettes
// On calcule
[xv,yv,zv]=eval3dp(vase,linspace(-%pi,%pi,20),1:16);

f=gcf();f.color_map=graycolormap(32);

plot3d(xv,yv,zv) //Dessin du vase
e1=gce();e1.color_mode = 24;e1.hiddencolor=30;
//XYZ est une matrice 3 x N dont les colonnes sont les
//    coordonnées des points représentant le vase.
XYZ=[xv(:)';yv(:)';zv(:)'];

//premiere transformation : translation + homothetie
XYZT=translation([1 3 -3],XYZ);
XYZH=homothetie([1 3 -3],1.5*[1 1 1],XYZT);

//dessin
plot3d(matrix(XYZH(1,:),4,-1),matrix(XYZH(2,:),4,-1),..
       matrix(XYZH(3,:),4,-1))
e2=gce();e2.color_mode = 24;e2.hiddencolor=30;

//deuxième transformation : translation + rotation
XYZR=rotation([40 0 90],XYZT);
//dessin
```

```
plot3d(matrix(XYZR(1,:),4,-1),matrix(XYZR(2,:),4,-1),..
       matrix(XYZR(3,:),4,-1))
e2=gce();e2.color_mode = 24;e2.hiddencolor=30;


//fixe les propriétés des axes
a=gca();a.data_bounds=[-3 -3 -3;3 3 3];
a.rotation_angles=[44 66];a.isoview='on';
```

Fichier source : `scilab/T3D.sce`

8.2 Équations linéaires

De nombreux problèmes peuvent se ramener à la résolution d'un système d'équations linéaires, c'est-à-dire par une relation $Ax = b$ où A est une matrice $(m \times n)$ connue, b un vecteur connu de taille n et x est le vecteur que l'on cherche.

Exemple : approximation polynomiale

Il est bien connu que l'équation du mouvement d'un corps en chute libre à la surface de la terre, si l'on ne veut pas tenir compte de la résistance de l'air, est de la forme suivante :

$$z(t) = -\frac{1}{2}gt^2 + v_0 t + z(0)$$

où $z(t)$ est la valeur de l'altitude en fonction du temps t, g est la constante de gravitation du lieu, v_0 est la vitesse initiale du corps et $z(0)$ est sa position initiale. $z(t)$ est donc décrit par un polynôme du second degré en la variable t.

Supposons que l'on ait mesuré lors d'une expérience les altitudes $z(t_1)$, $z(t_2)$, $\ldots$, $z(t_n)$ aux instants t_1, t_2, $\ldots$, t_n et que l'on souhaite estimer les paramètres inconnus (g, v_0 et $z(0)$) de l'équation du mouvement.

Les paramètres doivent vérifier les n équations :

$$z(t_i) = -\frac{1}{2}gt_i^2 + v_0 t_i + z(0) \qquad i = 1, \ldots, n \tag{8.1}$$

Il y a ici 3 inconnues à déterminer et trois mesures peuvent suffire. Mais pratiquement, compte tenu de l'imprécision des instruments de mesures, on préfère réaliser plus de mesures et ensuite trouver des paramètres permettant de représenter au mieux les données expérimentales.

Supposons que l'on ait mesuré :

$$\left[\begin{array}{l|cccccc} t(\mathrm{s}) & 1 & 3 & 5 & 9 & 15 & 20 \\ z(t)(\mathrm{m}) & 1996 & 1959 & 1882 & 1612 & 911 & 58 \end{array}\right]$$

À chaque mesure correspond une équation et les équations 8.1 ci-dessus peuvent s'écrire matriciellement sous la forme :

$$\begin{bmatrix} 1^2 & 1 & 1 \\ 3^2 & 3 & 1 \\ 5^2 & 5 & 1 \\ 9^2 & 9 & 1 \\ 15^2 & 15 & 1 \\ 20^2 & 20 & 1 \end{bmatrix} \begin{bmatrix} -\frac{g}{2} \\ v_0 \\ z(0) \end{bmatrix} = \begin{bmatrix} 1996 \\ 1959 \\ 1882 \\ 1612 \\ 911 \\ 58 \end{bmatrix}$$

C'est une équation de la forme $Ax = b$. Dans ce cas il y a plus d'équations que d'inconnues et le problème n'a pas (en général) de solution. Par contre il est possible de déterminer les inconnues x (ici les paramètres à estimer) pour minimiser la norme $||Ax - b||_2$ c'est-à-dire dans notre cas :

$$\sum_{i=1}^{n}(z(t_i) + \frac{1}{2}gt_i^2 - v_0 t_i - z(0))^2$$

Ce type de résolution s'appelle résolution aux moindres carrés. Pour résoudre le problème général :

$$\min ||Ax - b||_2$$

on écrit que le gradient de cette fonction par rapport à x est nul, ce qui donne :

$$A^T Ax = A^T b$$

La matrice $A^T A$ est une matrice carrée qui est (en général) inversible et l'équation précédente admet donc une solution unique $x = (A^T A)^{-1} A^T b$. Pour inverser une matrice, la fonction standard est `inv`. On pourrait donc obtenir la solution avec la commande `inv(A'*A)*A'*b`. Cependant cela peut être assez coûteux sur des problèmes de grande taille, car cela implique que l'on calcule d'abord la matrice inverse `inv(A'*A)`, puis le produit de cette matrice par le vecteur b. L'opération \ (backslash) permet de calculer directement la solution d'un système d'équations linéaires, sans passer par l'opération intermédiaire de l'inversion de la matrice. On pourrait donc utiliser ici la commande `(A'*A)\A'*b`.

Mais l'opération backslash \ peut aussi s'appliquer à une matrice non carrée. Le résultat est la solution au sens des moindres carrés, lorsque du moins cette solution est bien définie de manière unique, ce qui est en général le cas lorsque l'on a plus d'équations que d'inconnues. Cela permet de réduire encore le calcul des inconnues, en particulier en évitant le produit `A'*A`. Le script suivant illustre ces calculs.

```
//Les mesures
t=[1;3;5;9;15;20];z=[1998;1958;1881;1613;911;58];
//les matrices A et b
A=[t.^2, t, ones(t)];b=z;
//Résolution du problème aux moindres carrés
x=A\b;
```

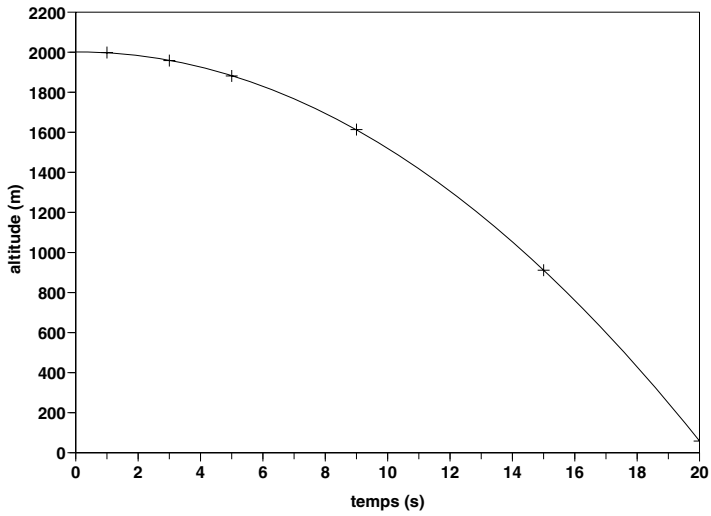


Figure 8.2 – Estimation de paramètres (exemple de la chute libre).

```
g=-2*x(1);v0=x(2);z0=x(3);
mprintf(" Gravité \t\t= %.3f m/s^2\n "..
+" Vitesse initiale \t= %.3f m/s\n"..
+" Altitude initiale \t= %.3f m\n",g,v0,z0)

//Génération de la figure
//-Affichage des mesures (croix)
plot2d(t,z,style=-1)
//-Affichage de la trajectoire estimée
t1=linspace(0,20,30);
plot2d(t1,-0.5*g*t1.^2+v0*t1+z0)
xtitle("","temps (s)","altitude (m)")
```

Fichier source : `scilab/chute1.sce`

On obtient :

```
 Gravité               = 9.796 m/s^2
 Vitesse initiale      = 0.828 m/s
 Altitude initiale     = 2000.752 m
```

et la figure 8.2 où l'on observe la trajectoire du corps correspondant aux paramètres estimés ainsi que les points effectivement mesurés.

Plus généralement il est de la même façon possible d'estimer les coefficients d'un polynôme de degré n. Chaque ligne de la matrice A et du vecteur b est

associée à une observation (x_i, y_i). Une ligne de A est formée des puissances successives de x_i de 0 jusqu'au degré du polynôme. Ce type de matrice s'appelle matrice de Vandermonde.

Si `X` et `Y` sont les vecteurs colonnes des observations, la solution `C` du problème aux moindres carrés peut être calculée par la fonction ci-dessous :

```
function p = polyfit(x,y,n)
  x = x(:); y = y(:);// vecteurs colonnes
  //normalisation et centrage de x pour rendre la matrice
  //Vandermonde mieux conditionnée
  m = mean(x); e = st_deviation(x);
  x = (x - m)/e;
  //Construction de la matrice de Vandermonde
  V = (x*ones(1,n+1)) .^ (ones(x)*(0:n))
  //Resolution
  p = poly(V\y,'x','c')
  //Changement de variable x=(x-m)/e
  p = horner(p,poly([-m,1]/e,'x','c'))
endfunction
```

Fichier source : `scilab/polyfit.sci`

Le script suivant réalise alors l'estimation aux degrés 4 et 6 des polynômes approchant au mieux les données `X` et `Y`. La figure 8.3 visualise le résultat.

```
X=(1:10)';Y=[1;3;2;0;4;5;9;3;1;-5];// les données
plot2d(X,Y,-3)

//estimation à l'ordre 4
C4=polyfit(X,Y,4)
x=linspace(0,10,50); //Vecteur de discrétisation
y=horner(C4,x); //Évaluation du polynôme C aux points de x
plot2d(x,y,style=1);  // Visualisation

//Estimation à l'ordre 6
C6=polyfit(X,Y,6);
plot2d(x,horner(C6,x),2) // Visualisation
e=gce();e.children.line_style=2;//tiretés

legend(["Données";"Ordre 4";"Ordre 6"],4)
```

Fichier source : `scilab/polyfit.sce`

8.3 Équations non linéaires

Les systèmes d'équations non linéaires $f(x) = 0$ peuvent se résoudre par la fonction `fsolve`. L'utilisation de `fsolve` exige que le nombre d'équations soit égal au nombre d'inconnues.

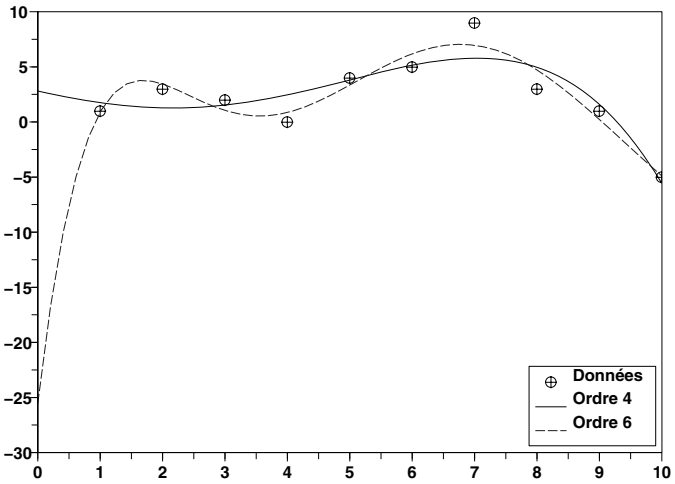


Figure 8.3 – Approximation polynomiale d'ordre 4 et 6.

Donnons un exemple scalaire. Sur le marché, le prix x d'un bien peut être défini par l'équation suivante :

$$x = \frac{1}{1+r}\left((1-p)x + p\left(x\int_0^x P(w)dw + \int_x^\infty wP(w)dw\right)\right)$$

où p est la probabilité d'obtenir une offre d'achat pour le bien sur une période donnée (par exemple une semaine), r est le taux d'intérêt sur cette période, et $P(w)$ est la densité de probabilité de la variable aléatoire w représentant la meilleure offre proposée sur cette même période.

Dans ce modèle on suppose que les offres non acceptées sont perdues. Le prix x correspond au seuil à partir duquel on accepte de vendre le bien dans une stratégie de maximisation du gain moyen. Le membre de droite représente l'espérance du gain futur actualisé, ce qui naturellement doit correspondre au prix actuel.

Cette équation représente une relation implicite en x, et il n'est pas possible d'exprimer explicitement x par des manipulations formelles ; pour la résoudre, on fait donc appel à `fsolve`. Bien entendu, dans cet exemple, la fonction `f` à fournir à `fsolve` s'obtient simplement en retranchant le membre droite du membre de gauche de l'équation précédente. La fonction `f` est donc définie comme suit :

```
function e=f(x)
```

```
//évaluation du résidu.
//xmin, xmax et r proviennent du contexte appelant
  e = x-(x*(1-p)+p*(x*intg(xmin,x,P)+intg(x,xmax,xP)))/(1+r)
endfunction


function y=P(w)
//fonction densité de probabilité: une loi gaussienne tronquée
//xmin, xmax, xmoyen, nrm et k proviennent du contexte appelant.
  if w<xmin|w>xmax then
    y=0
  else
    y=exp(-k*(w-xmoyen)^2)/nrm;
  end
endfunction

//fonction xP
function y=xP(w), y=w*P(w), endfunction
```

Fichier source : `scilab/prix.sci`

où `intg` est la primitive permettant de calculer l'intégrale d'une fonction (ici une fonction définie en langage Scilab, mais elle pourrait être définie en C ou fortran voir la section 7.4). Dans cet exemple, on prend comme densité de probabilité une loi gaussienne tronquée (fonction `P`).

Le script suivant permet alors de calculer le prix d'un bien en fonction du taux d'intérêt et de produire la figure 8.4 :

```
//Probabilité d'obtenir une offre/période :
p=0.5;
//Paramètres de densité en MegaEuros :
xmin=0.5;xmax=2;xmoyen=1;k=2;
//Normalisation de la densité :
nrm=1;nrm=intg(xmin,xmax,P);
//vecteur des taux :
R=linspace(0,0.1,100);

x=xmoyen; //initialization
X=[];
for r=R //boucle sur les taux
  [x,err]=fsolve(x,f);//Calcul du prix
  X=[X,x]; //memorisation
end

//Affichage
plot(R,X);   //Le prix en fonction du taux
w=linspace(0,2.5,500);
//densité de probabilité
```

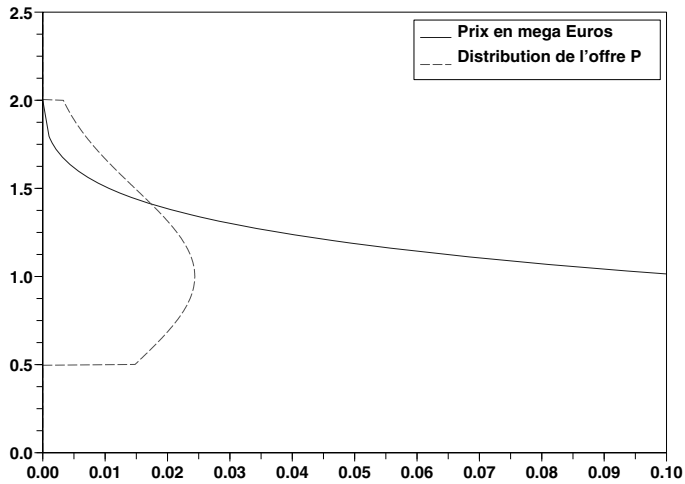


Figure 8.4 – Le prix en fonction du taux.

```
Z=[]; for www=w,  Z=[Z,P(www)]; end
plot(Z/40,w,'-r');  //Courbe de densité

legend(["Prix en mega Euros";
         "Distribution de l''offre P"]);
```

Fichier source : `scilab/prix.sce`

Noter que l'on utilise la valeur précédente du prix pour initialiser `fsolve` ; cela accélère de manière significative le calcul de `X`.

8.4 Optimisation

La fonction `optim` sert d'interface avec une importante bibliothèque de routines d'optimisation. En choisissant les paramètres de cette fonction, on peut utiliser diverses méthodes d'optimisation : quasi-newton, gradient conjugué et non différentiable.

Comme dans le cas de `fsolve`, l'utilisateur doit fournir comme paramètre d'entrée une fonction. Cette fonction Scilab doit retourner la valeur de la fonction à optimiser mais aussi le gradient de celle-ci, ce qui n'était qu'optionnel dans le cas de `fsolve`. Cette fonction peut être écrite en langage Scilab ou elle peut être une fonction externe écrite en C ou fortran (voir la section 7.4).

On donne ici un exemple simple d'utilisation de cette fonction dans un problème de gestion de stock. Soit x le stock courant d'un produit et u la quantité à commander, supposée positive ou nulle. On souhaite minimiser le coût J défini par :

$$J(u) = cu + \mathbb{E}\{\max(0, x + u - w)h + \max(0, w - x - u)p\}$$

où $\mathbb{E}$ désigne l'espérance mathématique et w est la variable aléatoire représentant le nombre de clients pour ce produit. h et p sont respectivement les coûts de stockage et le surcoût lié à la non satisfaction du client lorsque le produit n'est pas disponible. c est le coût de commande par unité de produit.

Soit $P(w)$ la densité de la variable aléatoire w, alors

$$J(u) = cu + \int_0^{+\infty} (h \max(0, x + u - w) + p \max(0, w - x - u))P(w)dw$$

et la dérivée de J par rapport à u est :

$$J_u(u) = c + h \int_0^{x+u} P(w)dw - p \int_{x+u}^{+\infty} P(w)dw$$

La fonction Scilab à fournir à `optim` peut s'écrire comme suit :

```
function [J,dJ,ind]=JJ(u,ind,  Big,c,x,p,h,k)
//Calcul du critére J et de son gradient dJ
  if (ind==2|ind==4) then //calcul du critère requis
    //le critère à minimiser J(u)
    J=c*u+intg(0,Big,list(fct,x,u,p,k))
  end
  if (ind==3|ind==4) then //calcul du gradient requis
    //La dérivée de J(u) par rapport à u
    dJ=c + intg(0,x+u,list(P,k))*h-intg(x+u,Big,list(P,k))*p
  end
endfunction


function z=fct(w,x,u,p,k)
//calcul de l'intégrande
// x, u, p et k sont fourni lors de l'appel a intg
  z=(max(0,x+u-w)*h+max(0,w-x-u)*p)*P(w,k)
endfunction

function y=P(w,k)
//Le nombre de clients : loi exponentielle
  y=k*exp(-k*w);
endfunction
```

Fichier source : `scilab/stock.sci`

Le membre de gauche de la fonction à minimiser `JJ` ainsi que les deux premiers arguments du membre de droite sont imposés, les autres arguments du membre de droite sont optionnels, leurs valeurs peut être fournies soit par le contexte, soit lors de l'appel à `optim` ; ici nous avons choisi la seconde méthode.

Noter que l'on a supposé que le nombre de clients w suivait une loi exponentielle.

Le script suivant dessine (figure 8.5) la quantité à commander optimale u en fonction du stock x.

```
//Les constantes du problème
Big=1000;//un grand nombre représentant l'infini
c=1;h=1;k=1/2;p=10;
stockmin=0; //borne inférieure pour le stock

S=linspace(0,10,100);
U=[];uopt=0;//initialisation
for s=S //calcul pour un stock variant entre 0 et 10
  //la fonction a minimiser et ses paramètres:
  Cost=list(JJ,Big,c,s,p,h,k);
  [cst,uopt]=optim(Cost,"b",stockmin,Big,uopt);
  U=[U,uopt];
end
plot(S,U)
xtitle("","stock (s)","commande (uopt)")
```

Fichier source : `scilab/stock.sce`

On constate que cette fonction est complètement caractérisée par un paramètre qu'on appelle seuil. Si x est en dessous de ce seuil s, la quantité à commander est $u = s - x$; dans le cas contraire $u = 0$. Le seuil s peut être obtenu par un seul appel à `optim` :

```
x=0; [cst,s]=optim(J,0)
```

8.4.1 Estimation de paramètres

Étant donné un ensemble d'observations f_i et v_i, $i = 1, \ldots, n$ correspondant respectivement à une force de résistance et une vitesse, on cherche des paramètres k et α pour que le modèle :

$$f_i = kv_i^\alpha, i = 1, \ldots, n$$

explique au mieux ces observations. Pour cela on minimise le critère suivant :

$$J(k, \alpha) = \sum_{i=1}^{n} \left(f_i - kv_i^\alpha\right)^2 .$$

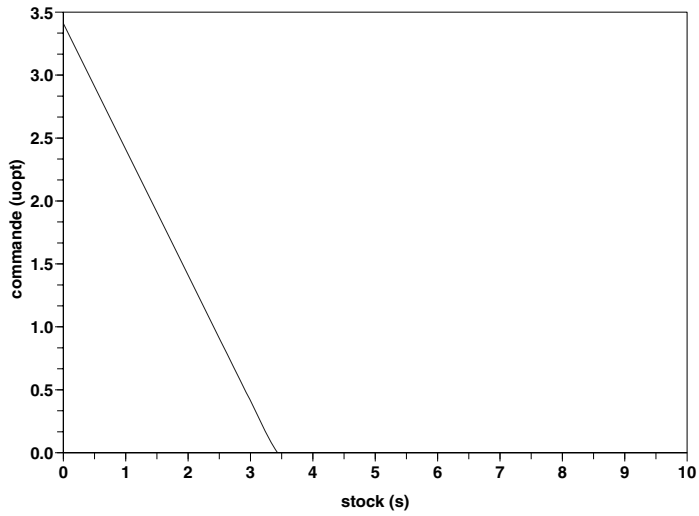


Figure 8.5 – La commande optimale en fonction du stock courant.

Dans ce problème, contrairement au cas précédent, on minimise par rapport à deux paramètres, donc le gradient sera ici un vecteur de taille 2 :

$$\begin{aligned} J_k(k,\alpha) &= -2\sum_{i=1}^{n}\left(f_i - kv_i^\alpha\right)v_i^\alpha \\ J_\alpha(k,\alpha) &= -2\sum_{i=1}^{n}\left(f_i - kv_i^\alpha\right)k\log(v_i)v_i^\alpha \end{aligned}$$

On peut facilement obtenir des données simulées pour ce problème :

```
k=3;alpha=2;n=50;v=rand(1,n);
f=k*v^alpha+0.2*rand(1,n,"n");
```

Fichier source : `scilab/estim.sce`

La fonction à minimiser et son gradient s'écrivent :

```
function [ct,gr,ind]=J(x,ind)
  //la valeur de f et v est fournie par le contexte
  k=x(1);alpha=x(2);   //x=[k,alpha]
  w=(f-k*v^alpha); //écarts
  ct=w*w';   //Valeur de J(x): somme des carrés des écarts
  gr=-2*[sum(w.*v^alpha),k*sum(w.*log(v).*v^alpha)]; //Gradient
endfunction
```

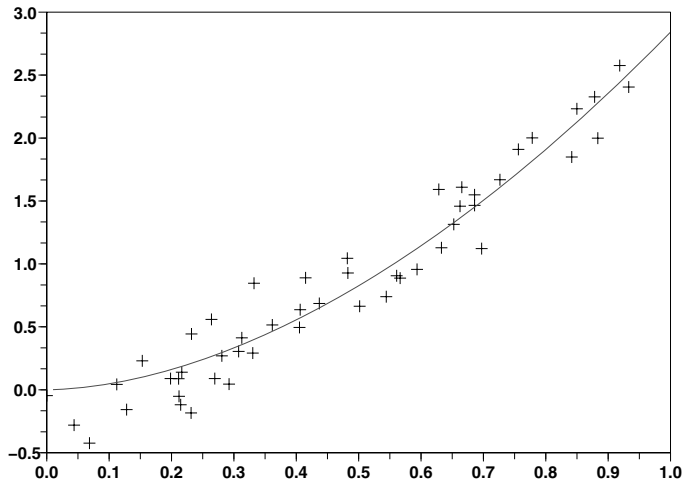


Figure 8.6 – Estimation de paramètres.

Fichier source : `scilab/estim.sci`

On peut alors faire appel à `optim` pour trouver la solution à partir d'une valeur initiale donnée et tracer (figure 8.6) les données ainsi que la courbe obtenue :

```
x0=[0 0]; // Valeur initiale de x=[k,alpha]
[copt,xopt]=optim(J,x0)

k=xopt(1); alpha=xopt(2);
plot2d(v,f,style=-1);
t=linspace(0.01,1,100);
plot2d(t,k*t^alpha,style=5)
```

Fichier source : `scilab/estim2.sce`

Ce même problème de moindres carrés non linéaires peut être résolu par le script suivant qui utilise la fonction `lsqrsolve`. Cette fonction implémente l'algorithme de Levenberg-Marquard. La fonction `E` ne calculant ici que le vecteur des écarts :

```
function w=E(x,m)
//l'argument m (dimension de w) n'est pas utilisé
```

```
//la valeur de f et v est fournie par le contexte
  k=x(1);alpha=x(2);   //x=[k,alpha]
  w=(f-k*v^alpha); //le vecteur des écarts
endfunction

m=size(v,'*');
lsqrsolve(x0,E,m)
```

Fichier source : `scilab/estim3.sce`

Si à la place de $(f_i - kv_i^\alpha)^2$, on avait pris comme critère $(\log(f_i) - \log(kv_i^\alpha))^2$, on aurait eu à résoudre un simple problème de moindres carrés linéaire, comme on en a vu précédemment au paragraphe 8.2, avec comme paramètres inconnus $\log(k)$ et α.

8.4.2 Une variante du problème de la chute libre

On considère de nouveau le problème de la chute libre mais cette fois on suppose que l'objet a été relâché à un instant inconnu t_0 avec une vitesse nulle et à une altitude inconnue y_0. Comme plus haut, on a à notre disposition des données temps-altitudes (t_i, y_i), $i = 1, \ldots, n$. En l'absence de bruit de mesure on aurait :

$$-\frac{1}{2}g(t_i - t_0)^2 + y_0 - y_i = 0 \qquad i = 1, \ldots, n \tag{8.2}$$

Les paramètres à estimer ici sont :

$$p = \begin{pmatrix} g \\ t_0 \\ y_0 \end{pmatrix}$$

et le vecteur de mesures est :

$$z_i = \begin{pmatrix} t_i \\ y_i \end{pmatrix}$$

On peut utiliser la fonction `optim` pour résoudre ce problème et pour cela il faut définir la fonction Scilab qui calcule la somme des carrés du membre de gauche de (8.2) ainsi que le gradient de celle-ci. La fonction `datafit` permet de résoudre ce type de problème, toujours en utilisant `optim`, mais avec un appel. En particulier, dans ce cas, il suffit de définir une matrice de données Z dont la i-ème colonne est z_i et une fonction Scilab $G(p, z)$ qui réalise le membre de gauche de (8.2), soit :

```
function e=G(p,z)
  e=-0.5*p(1)*(z(1)-p(2))^2+p(3)-z(2)
endfunction
```

L'appel à `datafit` se fait simplement en passant la fonction `G`, la matrice des données `Z` et une valeur initiale p_0. Le gradient de G n'est pas obligatoire, `datafit` l'estimant numériquement. S'il est fourni, la qualité et la vitesse de calcul sont supérieures. Avec les données du problème précédent on obtient :

```
->Z=[t';z'];
->p=datafit(G,Z,[0;0;1000])

 p  =

!   9.7959287 !
!   0.0845189 !
!   2000.7868 !
```

Chapitre 9

Systèmes d'équations différentielles

Les systèmes d'équations différentielles ([11]) sont utilisés dans la plupart des domaines du calcul scientifique. Ils permettent en particulier de modéliser les systèmes qui évoluent de façon continue avec le temps, systèmes que l'on appelle *systèmes dynamiques*.

Scilab dispose de solveurs permettant de résoudre numériquement de grandes classes de systèmes d'équations différentielles. Le choix du solveur dépend du type de l'équation. On peut distinguer deux grandes classes : les systèmes explicites et les systèmes implicites ou systèmes algébro-différentiels. La documentation sur ces solveurs pourra être trouvée dans les livres [3], [8], [9] et [15].

9.1 Systèmes explicites : le solveur `ode`

Scilab sait résoudre les systèmes d'équations différentielles explicites du premier ordre qui peuvent s'écrire :

$$\begin{cases} y' = f(t, y) \\ y(t_0) = y_0 \end{cases} \tag{9.1}$$

où y est le vecteur $(y_1, \ldots, y_n)^T$ qui dépend de la variable indépendante t, $f(t, y)$ est la fonction de plusieurs variables $(f_1(t, y), \ldots, f_n(t, y))^T$, t_0 est l'instant initial et y_0 est le vecteur des conditions initiales $y_0 = y(t_0)$. On a utilisé t pour représenter la variable indépendante car elle représente très souvent le temps et on a utilisé y' pour représenter la dérivée de y par rapport au temps.

9.1.1 Utilisation simple

Dimension 1, tracé de trajectoires

Nous traiterons ici le cas le plus simple d'équation différentielle, c'est-à-dire lorsque y dans (9.1) est de dimension 1 ; c'est par exemple le cas pour l'équation différentielle suivante :

$$\begin{cases} y' = y^2 - t \\ y(0) = 0 \end{cases} \tag{9.2}$$

On peut montrer que cette simple équation ne peut pas être résolue analytiquement, c'est-à-dire de façon exacte avec des fonctions connues, et donc seule une résolution numérique est possible.

C'est la fonction `ode` que l'on va utiliser pour résoudre cette équation, ici sous sa syntaxe la plus simple, c'est-à-dire :

```
y=ode(y0,t0,t,f)
```

où `y` est la solution qui est un vecteur de la taille de `t` : `y(i)` est la solution y au temps `t(i)`. On retrouve la condition initiale `y0` et l'instant initial `t0`. `t` est un vecteur qui représente les instants pour lesquels va être calculée la solution : son premier élément doit impérativement être `t0`. `f` est la fonction qui définit l'équation différentielle. Pour l'équation (9.2), cette fonction va être ici une fonction Scilab :

```
function yprim=f(t,y)
  yprim=y^2-t
endfunction
```

On va résoudre l'équation sur un intervalle de temps de 0 à 5, c'est-à-dire en partant du point $t = 0, y(0) = 0$ jusqu'à $t = 5$ par pas de temps de $0,1$. L'appel est alors le suivant :

```
t0=0; y0=0; tfin=5; dt=0.1; t=t0:dt:tfin; y=ode(y0,t0,t,f);
```

Il n'y a plus qu'à tracer la courbe de la solution en fonction du temps, c'est-à-dire la trajectoire, en utilisant la commande :

```
plot2d(t,y)
```

ce qui donne le tracé de la figure 9.1.

Cependant, très souvent, pour avoir une première idée de la forme des trajectoires, on trace les lignes de champ : ce sont les tangentes aux trajectoires pour tous les points d'une grille. Pour cela, on n'a pas besoin de résoudre l'équation et on utilise la fonction `champ`. On définit d'abord le cadre en (t, y), c'est-à-dire les valeurs minimales et maximales de t et y avec le pas de grille, dans lequel on va tracer les lignes de champ :

```
tmin=-3; tmax=5; dt=1; ymin=-3; ymax=3; dy=1; t=tmin:dt:tmax;
y=ymin:dy:ymax;
```

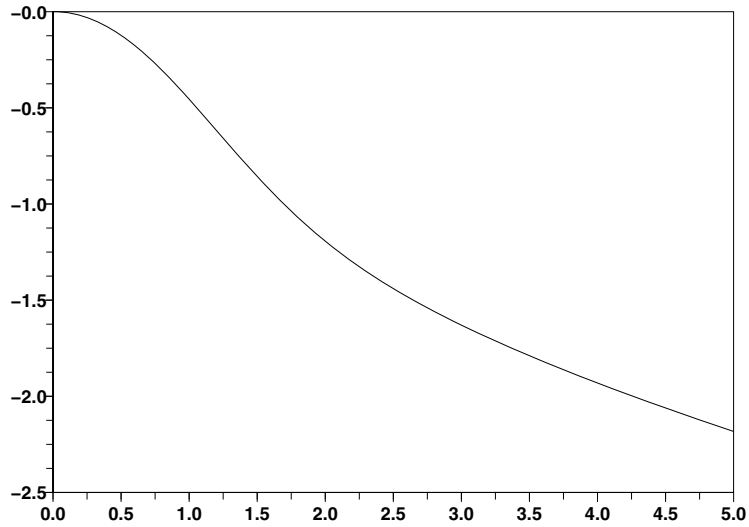


Figure 9.1 – Histogramme des données.

La fonction `champ` va tracer en chaque point de la grille (t_i, y_j) un petit vecteur de coordonnées $(c_t(t_i, y_j), c_y(t_i, y_j))$. Sa syntaxe est :

```
champ(t,y,ct,cy)
```

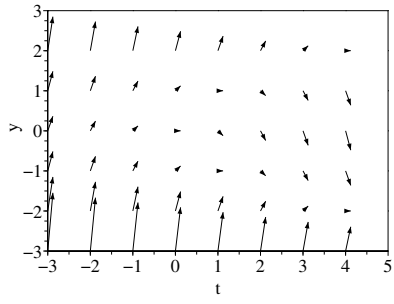
où les vecteurs `t` et `y` définissent la grille en t et en y et où les matrices `ct` et `cy` sont les valeurs du vecteur champ en chaque point de la grille. Comme l'on trace des tangentes aux trajectoires, le vecteur champ au point (t_i, y_j) est ici $(1, f(t_i, y_j))$. On obtient `cy`, c'est-à-dire ici $f(t_i, y_j)$ en utilisant la fonction `feval` qui évalue la fonction f aux points de la grille. On trace les lignes de champ :

```
// nombre de points de la grille
nt=size(t,"*"); ny=size(y,"*");
ct=ones(nt,ny);
cy=feval(t,y,f);
// on efface la fenêtre graphique
clf();
// tracé du champ de vecteurs
champ(t,y,ct,cy)
// tracé d'un titre
xtitle("","t","y")
// changement de l'épaisseur du cadre et des tracés
```

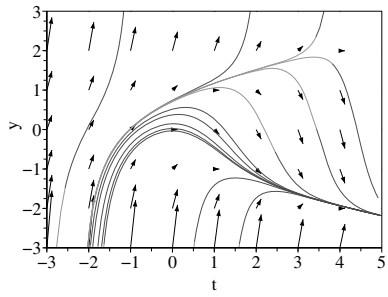
```
axe=gca(); axe.thickness=2;
// changement de la taille et du type des fontes
axe.font_style=2; axe.font_size=5;
axe.x_label.font_style=2; axe.x_label.font_size=5;
axe.y_label.font_style=2; axe.y_label.font_size=5;
```

Fichier source : `scilab/champ.sce`

Cela donne la figure 9.2(a).



(a)



(b)

Figure 9.2 – Lignes de champ et trajectoires pour $y' = y^2 - t$.

Il est maintenant intéressant de tracer des trajectoires dans la même grille que les lignes de champ. On peut utiliser simplement `ode` puis `plot2d` comme on l'a fait plus haut. On peut aussi cliquer sur un point du cadre et voir se tracer la trajectoire passant par ce point. Cela peut se faire en utilisant `xclick` décrite en section 5.6.1. Cette fonction attend le clic d'un bouton de la souris et retourne le numéro du bouton cliqué ainsi que les coordonnées du point cliqué. Pour tracer toutes les trajectoires que l'on désire à partir du point cliqué, il suffit donc de réaliser une boucle infinie jusqu'à ce que l'on clique, par exemple, sur le bouton droit de la souris (dans ce cas `c_i` dans le programme ci-dessous

est égal à 2). À partir de chaque point cliqué, on résout l'équation différentielle une fois dans le sens des t croissants jusqu'à `tmax` et une fois dans le sens des t décroissants jusqu'à `tmin`. Cela donne le code suivant :

```
t0=0; y0=0; dt=0.1;
// les tracés successifs utilisent le cadre du premier tracé
//  (celui  des lignes de champ fourni par le script champ.sce)
//  afin de réaliser une bonne superposition.
axe.auto_scale="off";
while(%t)
  [c_i,t0,y0]=xclick();
  // si le bouton droit est cliqué : arrêt
  if c_i==2 then break end;
  // test pour vérifier que l'on clique dans le cadre
  if t0>=tmin & t0<=tmax & y0>=ymin & y0<=ymax then
    // résolution de t0 vers tmax
    t=t0:dt:tmax; y=ode(y0,t0,t,f);
    plot2d(t(1:size(y,"*")),y,style=color("red"))
    // résolution de t0 vers tmin
    t=t0:-dt:tmin; y=ode(y0,t0,t,f);
    plot2d(t(1:size(y,"*")),y,style=color("green"))
  end
end
```

Fichier source : `scilab/trajectoires.sce`

Dans la fonction `plot2d`, on a utilisé `t(1:size(y,"*"))` car parfois la solution part à l'infini et le solveur s'arrête avant la fin. Donc on ne sait pas *a priori* combien de points ont été calculés ni quelle est la taille du résultat `y`. Cela donne la figure 9.2(b).

Dimension 2, tracés dans le plan de phase

Lorsque y est de dimension 2, c'est-à-dire lorsque l'on se trouve en présence d'un système de deux équations différentielles, on entre dans une grande classe de problèmes où il est possible de « voir » les solutions en se plaçant dans le *plan de phase*, c'est-à-dire le plan (y_1, y_2). Lorsque le système est *autonome*, c'est-à-dire lorsque le temps t n'apparaît pas explicitement dans f et que le système 9.1 s'écrit $y' = f(y)$, les trajectoires sont tracées comme des courbes dans ce plan de phase et sont appelées des *orbites*. On se déplace sur celles-ci lorsque le temps varie.

On va prendre l'exemple classique en dynamique des populations des équations de Lotka-Volterra, qui modélisent de façon simple le comportement du mélange de deux populations : les proies y_1 (les sardines) et les prédateurs y_2 (les requins). Le système s'écrit :

$$\left\{\begin{array}{ll} y_1' = a\,y_1 - b\,y_1 y_2 & a, b > 0 \\ y_2' = c\,y_1 y_2 - d\,y_2 & c, d > 0 \\ y_1(0) \quad \text{et} \quad y_2(0) & \text{donnés} \end{array}\right. \tag{9.3}$$

Ce modèle signifie qu'en l'absence de requins les sardines prolifèrent ($y_1' = a\,y_1$), qu'en l'absence de sardines les requins disparaissent ($y_2' = -d\,y_2$) et le terme en $y_1 y_2$, qui représente la rencontre des requins et des sardines, augmente le nombre de requins et diminue le nombre de sardines (car ces dernières sont mangées par les requins).

La fonction $f(t, y)$ est maintenant définie sous la forme d'une fonction vectorielle où y représente le vecteur (y_1, y_2) :

```
function yprim=lotka_volterra(t,y)
  yprim(1)=a*y(1)-b*y(1)*y(2);
  yprim(2)=c*y(1)*y(2)-d*y(2);
endfunction
```

Fichier source : `scilab/lotka_volterra.sci`

On va d'abord tracer les lignes de champ dans le plan de phase, ce qui permet à nouveau d'avoir une idée des orbites sans résoudre le système : cela revient à tracer le vecteur (y_1', y_2') en chaque point d'une grille. On utilise ici la fonction `fchamp` dont la syntaxe est :

```
fchamp(f,t,y1r,y2r)
```

où `f` est la fonction que l'on trace et `y1r` et `y2r` sont des vecteurs correspondant aux coordonnées des points de la grille respectivement en y_1 et en y_2. Ici t, instant choisi, ne sert à rien car le système d'équations est autonome (le temps n'intervient pas explicitement) et l'on mettra 0.

On choisit pour l'application numérique $a = 3$, $b = 1$, $c = 1$ et $d = 2$. On définit d'abord le cadre en (y_1, y_2), c'est-à-dire les valeurs minimales et maximales de y_1 et y_2 avec le pas de grille, dans lequel on va tracer les lignes de champ, puis on les trace :

```
//paramètres de l'équation
a=3;b=1;c=1;d=2;
// nombre de points de la grille
y1min=0; y1max=6; dy1=0.5;
y2min=0; y2max=6; dy2=0.5;
y1r=y1min:dy1:y1max; y2r=y2min:dy2:y2max;
// on efface la fenêtre graphique
clf();
fchamp(lotka_volterra,0,y1r,y2r)
// tracé d'un titre
xtitle("","y1","y2")
// changement de l'épaisseur du cadre et des tracés
axe=gca(); axe.thickness=2;
```

```
// changement de la taille et du type des fontes
axe.font_style=2; axe.font_size=5;
axe.x_label.font_style=2; axe.x_label.font_size=5;
axe.y_label.font_style=2; axe.y_label.font_size=5;
```

Fichier source : `scilab/lotka_volterra.sce`

Le champ de vecteurs est représenté sur la figure 9.3.

Comme dans le paragraphe précédent, on va maintenant tracer des orbites qui passent par le point cliqué :

```
// les tracés successifs utilisent le cadre du premier tracé
//  (celui des lignes de champ) afin de réaliser une bonne
//  superposition.
axe=gca();
axe.auto_scale="off";
// changement de l'épaisseur du trait
axe.thickness=2;
t0=0; tmax=5; dt=0.05;
t=t0:dt:tmax;
while(%t)
  [c_i,y1_0,y2_0]=xclick();
  // si le button droit est cliqué : arrêt
  if c_i==2 then break end;
  // test pour vérifier que l'on clique dans le cadre
  if y1_0>=y1min & y1_0<=y1max & y2_0>=y2min & y2_0<=y2max then
    y=ode([y1_0;y2_0],t0,t,lotka_volterra);
    plot2d(y(1,:),y(2,:),style=color("red"))
  end
end
```

Fichier source : `scilab/orbites.sce`

L'appel de `ode` est similaire à la dimension 1, si ce n'est qu'il faut donner un vecteur de conditions initiales et non plus une valeur scalaire. Ici `y` est une matrice à deux lignes dont chaque ligne correspond respectivement à y_1 et y_2. Les orbites tournent autour du point d'équilibre ($y_1 = d/c = 2, y_2 = a/b = 3$) (figure 9.3).

On peut aussi tracer une trajectoire correspondant à une solution du système. Cela revient à tracer une courbe en dimension 3 dans l'espace (y_1, y_2, t). Par exemple, on résout le système en partant du point $(1, 2)$ et on trace la courbe en utilisant la fonction `param3d`.

La suite de commandes :

```
y=ode([1;2],t0,t,lotka_volterra);
clf()
param3d(y(1,:),y(2,:),t)
axe=gca();
// on se donne le cadre
```

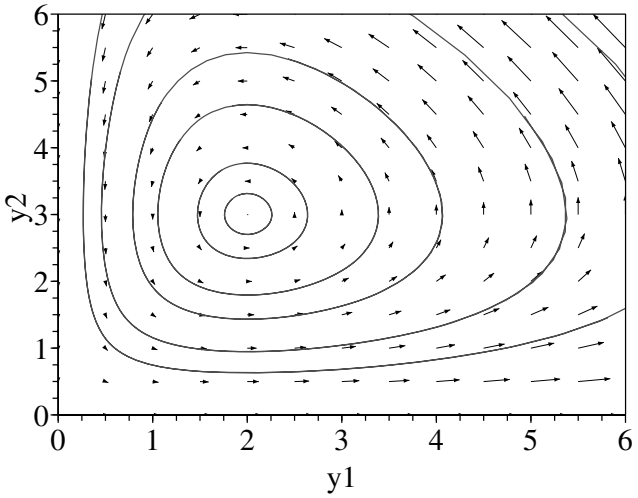


Figure 9.3 – Lignes de champ et orbites pour le système de Lotka-Volterra.

```
axe.data_bounds=[0 0 0;4 6 5];
// on met les labels des axes
axe.x_label.text="y1";
axe.y_label.text="y2";
axe.z_label.text="t";
```

Fichier source : `scilab/trajectoire3d.sce`

donne la figure 9.4. Comme le système est autonome, la trajectoire se projette en une courbe fermée sur le plan (y_1, y_2), autrement dit, toutes les trajectoires se déduisent les unes des autres par translation en temps.

Dimension quelconque

En dimension supérieure à 2 il devient difficile de réaliser des tracés de trajectoires car on se retrouve dans un espace $(t, y_1, \ldots, y_n)$ de dimension supérieure à 3. S'il est encore possible de réaliser des tracés de phase en dimension 3 dans l'espace (y_1, y_2, y_3) pour les systèmes de dimension 3, cela devient impossible au-delà. On tracera donc la variation de certaines dimensions par rapport au temps ou entre elles.

Pour la résolution de ces systèmes la syntaxe de `ode` reste bien sûr la même qu'en dimension 1 ou 2.

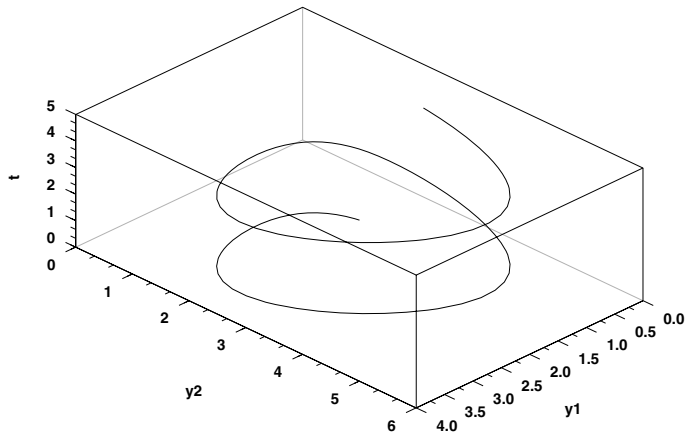


Figure 9.4 – Une trajectoire pour le système de Lotka-Volterra.

9.1.2 Temps d'arrêt

Parfois, on veut résoudre un système d'équations différentielles jusqu'à un temps que l'on ne connaît pas explicitement mais qui est donné par un comportement de la trajectoire. Par exemple, on veut résoudre le système de dimension n jusqu'à l'instant t où la trajectoire $y(t)$ (courbe dans l'espace $(t, y_1, \ldots, y_n)$ de dimension $n+1$) atteint ou traverse une surface (ou contrainte) $g(t, y) = 0$. Cet instant s'appelle un *temps d'arrêt*. Il peut aussi y avoir plusieurs surfaces $g_i(t, y) = 0$, auquel cas le temps d'arrêt est le temps auquel la première surface est atteinte.

Pour résoudre ce problème, on utilise une version adaptée du solveur `ode` dont la syntaxe la plus simple est :

```
y=ode("root",y0,t0,t,f,ng,g)
```

où l'on retrouve la condition initiale `y0` et l'instant initial `t0`, le vecteur `t` qui représente les instants pour lesquels la solution va être calculée, et la fonction `f` qui définit l'équation différentielle. En plus, `ng` est le nombre de surfaces et `g` est une fonction dont la syntaxe est `z=g(t,y)` : `z` est un vecteur de dimension `ng` dont chaque élément correspond à une valeur de $g_i(t, y)$, le but étant d'annuler un de ces éléments.

Si l'on reprend l'exemple du système (9.2) traité dans la page 183, on se souvient que la solution partait parfois à l'infini et donc que le solveur s'arrêtait prématurément avec des messages d'erreur. Une façon d'éviter ce comportement est de résoudre l'équation différentielle jusqu'aux bords du cadre en utilisant

l'option `"root"` de `ode`. Dans ce cas, les deux surfaces correspondent aux droites $g_1(t,y) = y+3$ (bord inférieur du cadre) et $g_1(t,y) = y-3$ (bord supérieur du cadre) définies simplement par la fonction :

```
function z=g(t,y),z=[y-3;y+3],endfunction
```

L'exemple s'écrit maintenant :

```
function z=g(t,y), z=[y-3;y+3],endfunction
t0=0; y0=0; dt=0.1;
a.auto_scale="off";
while(%t)
  [c_i,t0,y0]=xclick();
  // si le bouton droit est cliqué : arrêt
  if c_i==2 then break end;
  // test pour vérifier que l'on clique dans le cadre
  if t0>=tmin & t0<=tmax & y0>=ymin & y0<=ymax then
    // résolution de t0 vers tmax
    t=t0:dt:tmax;
    y=ode("root",y0,t0,t,lotka_volterra,2,g);
    plot2d(t(1:size(y,"*")),y,style=color("red"))
    // résolution de t0 vers tmin
    t=t0:-dt:tmin;
    y=ode("root",y0,t0,t,lotka_volterra,2,g);
    plot2d(t(1:size(y,"*")),y,style=color("green"))
  end
end
```

Fichier source : `scilab/trajectoires2.sce`

Ainsi les messages d'erreur disparaissent et la résolution est plus rapide.

Dans le cas précédent, on ne s'intéresse pas aux instants auxquels la trajectoire traverse la surface, mais très souvent ce temps fait partie de ce que l'on cherche : penser au temps d'atteinte d'une cible. On utilise alors la forme suivante de `ode` :

```
[y,rd]=ode("root",y0,t0,t,f,ng,g)
```

où `rd` est un vecteur de taille k. Son premier élément `rd(1)` est le temps d'arrêt. Les autres éléments indiquent quelles surfaces la trajectoire a traversées : par exemple, `[2.1,1,3]` indique que le temps d'arrêt est `2.1` et que les surfaces 1 et 3 ont été atteintes.

On peut utiliser ce calcul de temps d'arrêt dans le cadre du système de Lotka-Volterra (9.3) pour calculer par exemple la période de rotation autour du point d'équilibre. Pour cela, on va définir le plan passant par le point initial $(y_1(0), y_2(0))$ et perpendiculaire à la trajectoire et au plan de phase. On va ensuite résoudre le système à partir du point initial jusqu'à ce qu'il coupe 2 fois ce plan. On obtiendra ainsi la période (figure 9.5).

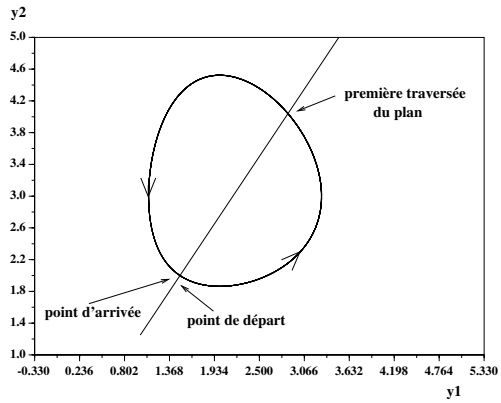


Figure 9.5 – Calcul de la période par la méthode du temps d'arrêt.

On définit d'abord la surface (le plan) $z = g(t, y)$:

```
function z=plan(t,y)
  v0=lotka_volterra(t0,Y0);
  z=(y(2)-Y0(2))*v0(2)+(y(1)-Y0(1))*v0(1);
endfunction
```

Fichier source : `scilab/plan.sci`

Puis on définit la fonction `periode`. La difficulté ici est que lorsque l'on se trouve sur la surface, il faut d'abord s'éloigner de la surface, sinon `ode` avec recherche de temps d'arrêt ne peut pas fonctionner (on est déjà arrivé !).

```
function T=periode(Y0)
// résolution sur un pas
  y1=ode(Y0,t0,[t0 t0+dt],lotka_volterra);
  // résolution jusqu'au plan
  [y2,rd]=ode("root",y1(:,2),t0+dt,[t0+dt:dt:tmax],..
                        lotka_volterra,1,plan);
  // instant de traversée du plan
  t1=rd(1);
  // résolution sur un pas
  y3=ode(y2(:,$),t1,[t1 t1+dt],lotka_volterra);
  // résolution jusqu'au plan
  [y,rd]=ode("root",y3(:,2),t1+dt,[t1+dt:dt:tmax],..
                         lotka_volterra,1,plan);
  // instant de retraversée du plan = instant cherché
  T=rd(1)
endfunction
```

Fichier source : `scilab/periode.sci`

Il ne reste plus qu'à réaliser le calcul à partir du point initial `[1.5;2]` par exemple. On obtient :

```
->a=3; b=1; c=1; d=2; ->t0=0; tmax=10; dt=0.001; ->t=t0:dt:tmax;
->Y0=[1.5;2]; ->T=periode(Y0)
 T  =
    2.6173305
```

9.1.3 Réglage du solveur

Dans les utilisations de `ode` que nous avons décrites jusqu'à présent, nous n'avons jamais influé sur le solveur lui-même. Il existe cependant un grand nombre de paramètres qu'il est possible de faire varier.

Les méthodes

Les solveurs utilisés par `ode` font partie du package ODEPACK[1] (programmes `lsode`, `lsoda` et `lsodar`). Ils permettent de résoudre à la fois les problèmes standard en utilisant une méthode de type Adams et les problèmes *raides*[2] en utilisant une méthode de type BDF. Le solveur reconnaît lui-même le type de problème auquel il a affaire. Il faut noter aussi que ces solveurs sont à pas et à ordre variables, c'est-à-dire que l'utilisateur spécifie les instants où il veut que la solution soit calculée, mais qu'en fait le solveur calcule de façon interne un pas variable optimal.

Ces solveurs admettent un grand nombre d'options plus ou moins complexes que l'on peut spécifier en utilisant la variable `%ODEOPTIONS` : consulter l'aide en ligne de `odeoptions`.

Les tolérances

Il est possible de jouer sur la vitesse de convergence du solveur et la précision du résultat à l'aide des tolérances d'erreurs relatives et absolues de l'intégration : ce sont les paramètres optionnels `rtol` et `atol` de `ode` . Ces paramètres prennent place après le temps `t` dans l'appel de `ode` qui devient :

```
y=ode(y0,t0,t,rtol,atol,f)
```

`rtol` et `atol` sont des scalaires ou des vecteurs de la même taille que le vecteur `y` et ils déterminent le contrôle d'erreur réalisé par le solveur sur chaque composante de `y` : l'erreur locale estimée pour `y(i)` doit être inférieure à `rtol(i)*abs(y(i))+atol(i)`. `rtol` et/ou `atol` peuvent être des scalaires, l'erreur étant alors la même pour toutes les composantes de `y`. Les valeurs par défaut de `rtol` et `atol` sont respectivement fixées à 10^{-7} et à 10^{-9} pour les solveurs standard de `ode`.

[1] Se reporter à `www.llnl.gov/CASC/odepack`.

[2] Un problème raide est un problème pour lequel les composantes de l'état peuvent avoir en certains points des ordres de grandeur très différents.

Ces deux paramètres sont importants dans le cas de problèmes raides ou difficiles à intégrer : il faut parfois leur donner des valeurs plus faibles que les valeurs par défaut. Un exemple dans le cas des systèmes implicites est donné plus loin, à la page 197.

Utilisation du jacobien

Pour des problèmes de grande taille, en particulier s'ils sont raides, il est possible de donner au solveur la valeur du jacobien du système afin d'accélérer la convergence. Le jacobien du système (9.1) de n équations différentielles est la matrice carrée $J(t, y)$ d'ordre n dont les éléments sont : $J_{ij}(t, y) = \partial f_i(t, y)/\partial y_j$. On obtiendrait par exemple pour le système de Lotka-Volterra (9.3) :

$$J(t, y) = \begin{pmatrix} a - b\,y_2 & -b\,y_1 \\ c\,y_2 & c\,y_1 - d \end{pmatrix}$$

Le jacobien est défini par une fonction `j=jac(t,y)`. On aurait pour le système de Lotka-Volterra :

```
function j=jac(t,y) j=[a-b*y(2)   -b*y(1);
     c*y(2) c*y(1)-d];
endfunction
```

et, après définition de cette fonction supplémentaire, l'appel du solveur devient :

```
y=ode([1;2],t0,t,f,jac);
```

Noter que l'on ne peut pas toujours calculer explicitement le jacobien, en particulier si la fonction qui donne le système est un programme C ou FORTRAN (voir 9.1.4).

9.1.4 Utilisation de C et FORTRAN

Jusqu'à présent on a toujours utilisé des fonctions Scilab pour définir la fonction $f(t, y)$ du système d'équations différentielles. Mais parfois le système n'est défini que par un programme écrit dans un langage qu'il faut compiler comme C, C++ ou FORTRAN. Parfois aussi, même s'il est possible de définir le système par une fonction Scilab, il peut être plus efficace d'utiliser une fonction C ou FORTRAN pour des raisons de rapidité de calcul : un tel exemple dans le cas des systèmes implicites est donné plus loin à la page 198.

On peut utiliser un sous-programme FORTRAN pour définir $f(t, y)$. Ses arguments doivent être la liste `(n,t,y,yprim)` où `n` est la dimension de y, `t` est le temps (utilisé pour les systèmes non autonomes), `y` est le vecteur y et `yprim` est la valeur de $f(t, y)$ et la sortie du sous programme. La forme générale de ce sous-programme est :

```
subroutine f(n,t,y,yprim)
double precision t,y(n),yprim(n)
...
end
```

On peut aussi utiliser une fonction C comme une procédure avec les mêmes arguments et la forme générale de cette fonction est :

```
f(int *n,double *t,double *y,double *yprim) { ... }
```

Noter que ces procédures seront directement appelées par la routine d'intégration d'équation différentielle qui impose la liste d'appel et le type des arguments.

Par exemple, pour le système de Lotka-Volterra (9.3), le sous-programme FORTRAN correspondant est :

```
      subroutine volterraf(n,t,y,yprim)
      double precision t,y(n),yprim(n)
      double precision a,b,c,d
      parameter (a=3,b=1,c=1,d=2)

      yprim(1) = a*y(1) - b*y(1)*y(2)
      yprim(2) = c*y(1)*y(2) - d*y(2)
      end
```

Fichier source : `C/volterraf.f`

et la fonction C correspondante est :

```
volterrac(int *n,double *t,double *y,double *yprim)
{
  double a = 3.0, b = 1.0, c = 1.0, d = 2.0;

  yprim[0] = a*y[0] - b*y[0]*y[1];
  yprim[1] = c*y[0]*y[1] - d*y[1];
}
```

Fichier source : `C/volterrac.c`

On remarque dans l'exemple précédent que l'on a défini les paramètres `a`, `b`, `c` et `d` dans le corps des programmes. Il est possible de les garder comme paramètres, en utilisant des fonctions fournies par l'environnement Scilab. Des exemples pour le passage général de paramètres sont donnés dans le répertoire

```
SCI/routines/examples/link-examples.
```

Il faut maintenant compiler le programme et le lier avec Scilab. La façon la plus simple est d'utiliser la méthode décrite dans la section 7.4.
Pour le FORTRAN :

```
mode(-1) //suppression de la visualisation
// Le chemin absolu du fichier
path = get_absolute_file_path("builder_vf.sce")
old=pwd();cd(path) //
nom_prog="volterraf";
```

```
fichier="volterraf.o";
libn=ilib_for_link(nom_prog,fichier,[],"f",..
   "../C/Makefile_vf","../C/loader_vf.sce");
cd(old); //retour dans le repertoire initial
clear old names path files flag
```

Fichier source : `C/builder_vf.sce`

et pour le C :

```
mode(-1) //suppression de la visualisation
// Le chemin absolu du fichier
path = get_absolute_file_path("builder_vc.sce")
old=pwd();cd(path) //
nom_prog="volterrac";
fichier="volterrac.o";
libn=ilib_for_link(nom_prog,fichier,[],"c",..
   "../C/Makefile_vc","../C/loader_vc.sce");
cd(old); //retour dans le repertoire initial
clear old names path files flag
```

Fichier source : `C/builder_vc.sce`

Une fois compilé on peut le charger avec le fichier "loader" généré et l'exécuter.
Pour le FORTRAN :

```
exec C/loader_vf.sce
//intégration de l'équation différentielle
t0=0; tmax=5; dt=0.05;
t=t0:dt:tmax;
y=ode([1;2],t0,t,"volterraf");
```

Fichier source : `scilab/volterraf.sce`

et pour le C :

```
exec C/loader_vc.sce
//intégration de l'équation différentielle
t0=0; tmax=5; dt=0.05;
t=t0:dt:tmax;
y=ode([1;2],t0,t,"volterrac");
```

Fichier source : `scilab/volterrac.sce`

Noter que le quatrième argument de `ode`, `"volterraf"` ou `"volterrac"`, est maintenant une chaîne de caractères car c'est une fonction externe que l'on appelle.

Il est bien sûr aussi possible de définir le jacobien du système (voir 9.1.3) par une fonction C ou FORTRAN, voir l'aide en ligne de `ode` pour cela.

9.2 Systèmes implicites : le solveur `dassl`

9.2.1 Utilisation simple

Nous n'avons traité jusqu'à présent que des systèmes explicites, c'est-à-dire pour lesquels les dérivées y' pouvaient s'écrire explicitement en fonctions de y et de t. On ne se trouve pas toujours dans un cas aussi simple. Les problèmes implicites sont très difficiles voire impossibles à résoudre. Le solveur `dassl` permet de résoudre un certain nombre de systèmes implicites de la forme :

$$\begin{cases} f(t, y, y') = 0 \\ y(t_0) = y_0 \end{cases} \tag{9.4}$$

On parle souvent aussi de *systèmes algébro-différentiels* qui combinent un système explicite avec un système algébrique comme, par exemple :

$$\begin{cases} y' = f(t, y, z) \\ 0 = g(t, y, z) \end{cases}$$

Cependant de tels systèmes se ramènent au système (9.4) et de toute façon `dassl` résout le système sous la forme (9.4).

`dassl` sait résoudre les systèmes d'indice 1 et quelques systèmes d'indice 2. L'indice représente formellement le nombre de dérivations qui permettent de rendre explicite un système implicite, un système explicite étant d'indice 0.

C'est donc la fonction `dassl` que l'on va utiliser pour résoudre ces systèmes implicites. Sa syntaxe la plus simple est :

```
y=dassl([y0,yprim0],t0,t,f)
```

où, comme pour `ode`, `y0` est la condition initiale sous la forme d'un vecteur *colonne* et `t0` est l'instant initial. `t` est un vecteur qui représente les instants pour lesquels va être calculée la solution : son premier élément doit impérativement être `t0`. Ici `y` est une matrice dont chaque colonne i est le vecteur

$$(t_i y_1(t_i) \dots y_n(t_i) y_1'(t_i) \dots y_n(t_i))$$

où n est la dimension de y. En revanche, `dassl` a besoin d'une donnée supplémentaire qui est la valeur de la dérivée de y à l'instant initial $y'(t_0)$ qui est donnée sous la forme du vecteur *colonne* `yprim0`. Enfin, `f` est la fonction qui définit le système et doit être de la forme `[r,ir]=f(t,y,yprim)` où `ir` est un drapeau de retour qui doit valoir 0 quand le calcul de f s'est bien passé, -1 si la valeur de f n'est pas définie et -2 si les arguments de la fonction ont des valeurs non admissibles.

Il faut noter qu'il est nécessaire que les données initiales soient consistantes, c'est-à-dire que l'on doit avoir `f(t0,y0,yprim0)=0`. Il n'est pas toujours évident de trouver une valeur pour `yprim0` et il faut parfois utiliser `fsolve` pour cela (voir la section 8.3).

Ensuite, il est possible, comme pour `ode`, de tracer des lignes de champ et des trajectoires en utilisant les mêmes fonctions de tracé.

9.2.2 Utilisation avancée

Toutes les fonctionnalités du solveur `ode` sont disponibles aussi pour le solveur `dassl` :

- Il est possible de calculer des temps d'arrêt (voir 9.1.2) en utilisant la fonction `dasrt`.
- Il est possible de régler le solveur, aussi bien au niveau des tolérances (voir 9.1.3) qu'au niveau de la méthode en utilisant l'argument optionnel `info` de `dassl`.
- Il est possible de fournir le jacobien du système (voir 9.1.3) pour les problèmes raides.
- Enfin on peut utiliser des programmes C et FORTRAN (voir 9.1.4) pour définir le système et/ou le jacobien. Des exemples pour l'utilisation de `dassl` et `dasrt` avec FORTRAN sont donnés dans les fichiers `Ex-dassl.f` et `Ex-dassrt.f` du répertoire `SCI/routines/default`. Voir aussi la page 198.

9.2.3 Un exemple en mécanique

Il n'est pas facile de trouver des problèmes implicites simples que l'on peut résoudre sans trop de calculs. Un exemple pas trop compliqué est celui du pendule qui glisse avec frottement le long de la parabole $y = x^2$ dans un plan. Voir la figure 9.6.

Nous donnons ci-dessous les équations du mouvement du pendule qui vont nous donner le système différentiel à résoudre :

$$\begin{cases} x' = u_1 \qquad y' = u_2 \qquad \theta' = u_3 \\ (M+m)\,u_1' + ml\cos(\theta)\,u_3' = ml\sin(\theta)u_3^2 - 2\lambda x - ku_1 \\ (M+m)\,u_2' + ml\sin(\theta)\,u_3' = -ml\cos(\theta)u_3^2 - (M+m)g - \lambda - ku_2 \\ ml\cos(\theta)\,u_1' + ml\sin(\theta)\,u_2' + ml^2\,u_3' = -mg\sin(\theta) \\ 0 = -2xu_1 + u_2 \end{cases}$$

où x et y sont les coordonnées du bout supérieur du pendule, θ est l'angle que fait le pendule avec la verticale, M est la masse de l'extrémité supérieure du pendule, m celle de l'extrémité inférieure, l est la longueur du pendule, g est l'accélération de la pesanteur, k est le coefficient de frottement et λ est un multiplicateur de Lagrange. Pour obtenir ce système, nous avons utilisé une formulation de Lagrange avec la contrainte pour le bout supérieur du pendule de suivre la courbe tout en introduisant un frottement proportionnel à la vitesse au niveau du glissement sur la courbe (fonction de dissipation de Rayleigh).

Ce système est d'indice 3 et `dassl` ne peut pas le résoudre, donc on a différencié la contrainte une fois afin d'obtenir un système d'indice 2.

Pour utiliser la fonction `dassl`, on définit d'abord le système différentiel précédent sous la forme $f(t, Y, Y') = 0$. La fonction `f` correspondante est :

```
function [r,ir]=pendule(t,Y,Yprim)
  x=Y(1); y=Y(2); theta=Y(3); u=Y(4:6); lambda=Y(7);
```

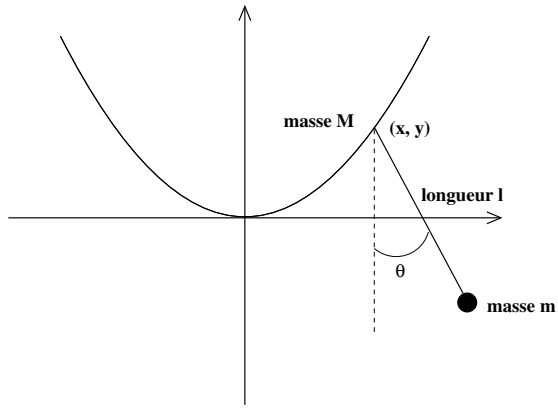


Figure 9.6 – Le pendule qui glisse le long d'une parabole.

```
  xprim=Yprim(1); yprim=Yprim(2); thetaprim=Yprim(3);
  uprim=Yprim(4:6);
  r(1)=xprim-u(1);
  r(2)=yprim-u(2);
  r(3)=thetaprim-u(3);
  r(4)=(M+m)*uprim(1)+m*l*cos(theta)*uprim(3)..
       -m*l*sin(theta)*u(3)^2-2*lambda*x+k*u(1);
  r(5)=(M+m)*uprim(2)+m*l*sin(theta)*uprim(3)..
       +m*l*cos(theta)*u(3)^2+(M+m)*g+lambda+k*u(2);
  r(6)=m*l*cos(theta)*uprim(1)+m*l*sin(theta)*uprim(2)+..
       m*l^2*uprim(3)+m*g*sin(theta);
  r(7)=-(-2*x*u(1)+u(2));
  ir=0
endfunction
```

Fichier source : `scilab/pendule.sci`

où `Y` est ici un vecteur de 7 éléments : par exemple `Y(1)` et `Y(2)` sont les coordonnées x et y du bout supérieur du pendule et `Y(3)` est l'angle θ qu'il fait avec la verticale. On appelle alors `dassl` :

```
// les données numériques
g=10; l=1; m=1; M=1; k=0.5;
// les valeurs initiales
x0=1; y0=1; theta0=0; u0=[0;0;0];
Y0=[x0;y0;theta0;u0;0];
uprim0=[0;-g;0]; Yprim0=[u0;uprim0;0];
// instants où l'on calcule
t0=0; t=t0:0.05:20;
```

```
// paramètres de dassl
atol=[0.0001;0.0001;0.0001;0.0001;0.0001;0.0001;0.001];
rtol=atol;
// appel de dassl
Y=dassl([Y0,Yprim0],t0,t,rtol,atol,pendule);
```

Fichier source : `scilab/pendule.sce`

On remarque que l'on a utilisé les arguments `rtol` et `atol` afin d'augmenter leurs valeurs par défaut. En particulier la valeur `0.001` correspond au multiplicateur de Lagrange dont la précision du calcul est moins importante. Le problème à résoudre étant assez complexe, si l'on laisse les valeurs par défaut de ces paramètres, le solveur peut ne pas converger.

Le résultat se retrouve dans `Y` et il faut savoir que l'on retrouve la position du bout supérieur du pendule $(x(t), y(t))$ dans `Y(2,:)` et `Y(3,:)` et l'angle θ que fait le pendule avec la verticale se retrouve dans `Y(4,:)`.

Maintenant, nous allons réaliser l'animation du pendule. Pour cela on définit d'abord une fonction qui va définir toutes les entités graphiques dont on a besoin et qui va retourner son handler :

```
function  H=contruit_pendule()
// construit la figure du pendule avec ses entités graphiques
// retourne un handle sur les parties qui bougent
  clf();
  // définition de la figure
  f=gcf();a=gca();
  a.isoview="on";
  f.pixmap='on';
  a.box="off";

  xmin=-1.5; xmax=1.5; ymin=-1.1; ymax=2.35
  a.data_bounds=[xmin ymin;xmax ymax]
  // le cadre
  xrect(xmin,ymax,xmax-xmin,ymax-ymin)
  // la courbe
  vx=[xmin:0.01:xmax]'; vy=vx.*vx;
  xpoly(vx,vy,"lines")
  c=gce();c.foreground=color("red");
  // la tige du pendule
  x=0; y=0; teta=0;
  xp=x+l*sin(teta); yp=y-l*cos(teta);
  r=0.05 // le demi diamètre de la boule du pendule
  xp1=x+(l-r)*sin(teta); yp1=y-(l-r)*cos(teta);
  xpoly([x;xp1],[y;yp1],"lines")
  p=gce();p.thickness=2;
  // la boule du pendule
  xfarc(xp-r,yp+r,2*r,2*r,0,360*64)
  b=gce()
```

```
  H=glue([p,b]) //retourne le handle sur la tige et la boule
endfunction
```

Fichier source : `scilab/construit_pendule.sci`

On définit ensuite une fonction qui va dessiner le pendule en une position donnée :

```
function  dessine_pendule(H,position)
// dessine une position du pendule
  x=position(1); y=position(2); teta=position(3);
  b = H.children(1);r=b.data(3)/2
  xp=x+l*sin(teta); yp=y-l*cos(teta);
  xp1=x+(l-r)*sin(teta); yp1=y-(l-r)*cos(teta);
  p = H.children(2);p.data=[x, y; xp1, yp1];
  b = H.children(1); b.data=[xp-r,yp+r,2*r,2*r,0,360*64];
  show_pixmap()
endfunction
```

Fichier source : `scilab/dessine_pendule.sci`

Dans les fonctions ci-dessus, la propriété `f.pixmap='on';` permet de ne pas dessiner sur l'écran mais en mémoire. Pour obtenir l'affichage sur l'écran, il suffit de faire `show_pixmap()`.

Le script suivant permet alors de visualiser les résultat de la simulation sous forme d'une animation :

```
//On dessine le pendule dans sa position initiale :
H=contruit_pendule();
dessine_pendule(H,Y0(1:3));

//on dessine les positions successives du pendule :
realtimeinit(0.08);realtime(0)
for i=1:size(Y,2)
  realtime(i);
  dessine_pendule(H,Y(2:4,i))
end
```

Fichier source : `scilab/anim_pendule.sce`

où `realtime` permet de définir l'unité de temps et de synchroniser la simulation avec le temps.

Amélioration du temps de calcul

Supposons que nous voulions calculer la solution pour un grand nombre d'instants, par exemple 4000 en faisant :

```
t0=0; t=t0:0.05:200;
```

Alors le temps mis par `dassl` pour calculer la solution est de 11 secondes sur notre machine de référence :

```
->timer();Y=dassl([Y0,Yprim0],t0,t,rtol,atol,pendule);timer()
 ans  =
     2.11
```

La fonction f du système est déjà assez complexe et on peut essayer d'améliorer les performances de `dassl` en utilisant par exemple une fonction C pour la définir au lieu d'une fonction Scilab. La fonction C s'écrit :

```
#include <math.h>

pendulec(double *t,double *yy,double *yyprim,double *r,int *ir,
  double *rpar, int *ipar)
{
  double x,y,theta,xprim,yprim,thetaprim;
  double u[3],uprim[3],lambda;
  double g,k,l,gm,m;

  g=10;
  k=0.5;
  l=1;
  gm=1;
  m=1;

  x=yy[0];
  y=yy[1];
  theta=yy[2];
  u[0]=yy[3];
  u[1]=yy[4];
  u[2]=yy[5];
  lambda=yy[6];
  xprim=yyprim[0];
  yprim=yyprim[1];
  thetaprim=yyprim[2];
  uprim[0]=yyprim[3];
  uprim[1]=yyprim[4];
  uprim[2]=yyprim[5];
  r[0]=xprim-u[0];
  r[1]=yprim-u[1];
  r[2]=thetaprim-u[2];
  r[3]=(gm+m)*uprim[0]+m*l*cos(theta)*uprim[2]-m*l*sin(theta)
    *u[2]*u[2]-2*lambda*x+k*u[0];
  r[4]=(gm+m)*uprim[1]+m*l*sin(theta)*uprim[2]+m*l*cos(theta)
    *u[2]*u[2]+(gm+m)*g+lambda+k*u[1];
  r[5]=m*l*cos(theta)*uprim[0]+m*l*sin(theta)*uprim[1]+
    m*l*l*uprim[2]+m*g*sin(theta);
```

```
  r[6]=-(-2*x*u[0]+u[1]);
  ir=0;
}
```

Fichier source : `C/pendule.c`

Remarquer à nouveau l'utilisation des pointeurs pour passer les arguments de la fonction. On compile cette fonction avec `ilib_for_link`

```
mode(-1) //suppression de la visualisation
// Le chemin absolu du fichier
path = get_absolute_file_path("builder_pendule.sce")
old=pwd();cd(path) //
nom_prog="pendulec";
fichier="pendule.o";
libn=ilib_for_link(nom_prog,fichier,[],"c",..
   "Makefile_pend","loader_pendule.sce");
cd(old);
```

Fichier source : `C/builder_pendule.sce`

et on la charge en utilisant `link` :

```
->exec C/loader_pendule.sce; ->timer();
->Y=dassl([Y0,Yprim0],t0,t,rtol,atol,"pendulec");timer()
 ans  =
    0.04
```

Pour préciser que l'on utilise une fonction externe, on a utilisé une chaîne de caractères `"pendulec"` dans l'appel de `dassl`.

Lorsque le système que l'on veut traiter commence à devenir complexe, il est donc avantageux de le définir en C ou en FORTRAN au lieu d'utiliser une fonction Scilab.

Chapitre 10

Scicos

Comme on l'a vu dans le chapitre précédent, il est possible de construire et de simuler des modèles de systèmes dynamiques dans Scilab en utilisant les fonctions de base comme `ode` (solveur de systèmes d'équations différentielles). Mais pour cela, il faut représenter le modèle du système à simuler sous forme d'un programme Scilab, C ou FORTRAN. De plus, si le système contient des composants discrets et événementiels, l'utilisation de Scilab nécessite en général beaucoup de programmation complexe, difficile à déboguer et trop particulière pour conduire à du code réutilisable.

Scicos est une boîte à outils fournissant un éditeur graphique de type schéma-blocs qui permet de construire ces modèles en reliant des composants ou blocs représentant des petits systèmes dynamiques, par des liens qui représentent des flux d'information.

L'utilisateur peut décrire son système dynamique de façon complètement modulaire. Le modèle ainsi obtenu peut alors être compilé, et le résultat peut être simulé par le simulateur Scicos.

Les systèmes modélisables et simulables par Scicos sont de nature très variée ; ils peuvent comporter des composants évoluant en temps continu, en temps discret, et même des composants événementiels. On les appelle des systèmes hybrides. On présente ici l'utilisation de Scicos à travers plusieurs exemples.

10.1 Exemple simple

Supposons que l'on veuille modéliser un simple système dynamique dont la sortie est l'intégrale du sinus de son entrée. Pour pouvoir simuler le fonctionnement de ce système, en plus de l'intégrateur et du sinus, on a besoin d'un bloc pour générer une entrée test et un bloc d'affichage graphique pour visualiser la sortie.

10.1.1 Présentation de la fenêtre d'édition

On commence par lancer l'éditeur Scicos (dans la fenêtre Scilab) avec la commande :

```
-> scicos();
```

Cela ouvre la fenêtre principale de l'éditeur Scicos, contenant un schéma vide. La fenêtre et le schéma correspondant sont nommés par défaut `Untitled`.

Les fonctionnalités de Scicos sont disponibles à travers des menus déroulants. On devine assez facilement la fonction de chaque bouton de chaque menu par son nom : `Save` pour enregistrer un schéma, `Load` pour le charger, `Copy` pour copier un bloc, `Delete` pour le supprimer, `Color` pour donner une couleur au bloc, etc. Certaines fonctions sont aussi accessibles par des raccourcis clavier. Les raccourcis clavier sont définis par la variable Scilab `%scicos_short` et, dans Scicos, ils peuvent être consultés de modifiés en cliquant sur le bouton `Shortcuts` du menu `Misc`.

Il existe aussi un bouton `Help` qui se trouve dans le menu `Misc` qui permet de consulter l'aide en ligne sur chaque menu. Pour cela, il suffit de cliquer sur `Help` puis sur le bouton *ad hoc*. Le bouton `Help` permet aussi d'accéder à l'aide sur les blocs ; il suffit de cliquer sur un bloc après avoir cliqué sur `Help`.

10.1.2 Blocs et palettes

On commence par chercher les blocs dont on a besoin dans les palettes Scicos. Une palette est un schéma contenant des blocs prédéfinis que l'on peut utiliser pour construire d'autres schémas. Pour ouvrir une palette, il faut cliquer sur le bouton `Palette` du menu `Edit`. On voit alors une liste de palettes ; choisissons la palette `Linear`. On voit s'ouvrir une fenêtre graphique dont le contenu est représenté en figure 10.1.

Le label du bloc intégrateur est `1/s`. Pour copier ce bloc dans le schéma `Untitled`, il suffit de placer la souris sur ce bloc et de cliquer sur le bouton gauche de la souris, puis placer la souris dans la fenêtre qui contient le schéma `Untitled` à l'endroit où l'on veut placer le bloc, et cliquer à nouveau. Une copie du bloc `1/s` apparaît alors dans `Untitled`.

La fonction sinus est réalisée par le bloc `Trig.Function` qui est disponible dans la palette `Non_linear`. On copie ce bloc dans le schéma comme nous l'avons fait pour `1/s`. Pour connecter la sortie de `Trig.Function` à l'entrée du bloc `1/s`, on clique d'abord près du port de sortie de `Trig.Function` puis près du port d'entrée de `1/s` (voir la figure 10.2 qui correspond au fichier source *scicos_ diag/scicos200.cos*).

Il faut maintenant choisir une entrée : par exemple t (le temps). Pour cela, on va chercher dans la palette `Sources` le générateur de temps (l'horloge noire ayant une sortie à droite) et on connecte sa sortie à l'entrée du bloc `Trig.Function`. Le schéma ainsi obtenu est simulable mais pas intéressant à simuler car il n'y a pas d'affichage ; il faut rajouter de quoi visualiser le résultat. On place donc le bloc

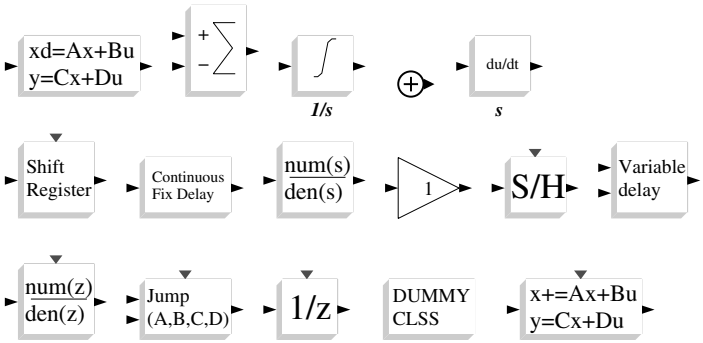


Figure 10.1 – La palette `Linear`.

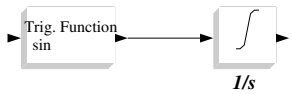


Figure 10.2 – Schéma Scicos incomplet.

`MScope` (oscilloscope à entrées multiples) de la palette `Sinks` dans le schéma et on connecte la sortie de `1/s` à sa première entrée et la sortie de `Trig.Function` sur sa deuxième entrée. Comme la sortie de `Trig.Function` est déjà branchée, il faut placer la souris quelque part sur le lien qui sort de `Trig.Function` et cliquer, ce qui crée un branchement et débute un lien, puis cliquer sur l'entrée de `MScope`. Pour obtenir un schéma plus lisible, il est préférable de ne pas créer dans ce dernier cas un lien direct qui passerait par dessus le bloc `1/s`, mais de le contourner. Pour cela, après avoir débuté le lien, on peut cliquer sur des points intermédiaires avant de cliquer sur l'entrée de `MScope`.

Le bloc `MScope` est un bloc discret qui affiche les valeurs de ses entrées quand il est activé. Pour activer ce bloc, il faut envoyer un événement sur son port d'entrée d'activation placé en haut du bloc. Dans ce cas on veut envoyer une suite d'événements espacés de manière régulière dans le temps. On utilise donc le bloc horloge d'événements (l'horloge rouge ayant une sortie en bas qui se trouve dans la palette `Sources`). Une fois ce bloc placé et branché, le schéma complet doit ressembler a celui de la figure 10.3 qui correspond au fichier source *scicos_ diag/scicos3000.cos*.

10.1.3 Simuler un schéma

Un schéma peut être simulé en cliquant sur le bouton `Run` du menu `Simulate`. On observe alors l'ouverture d'une fenêtre graphique fonctionnant comme un

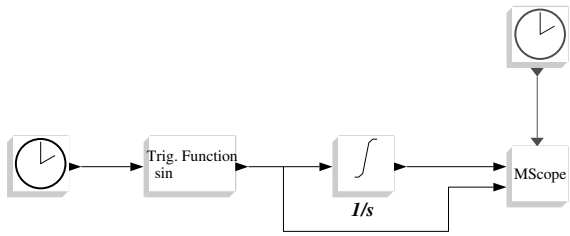


Figure 10.3 – Schéma complet.

oscilloscope. Pour arrêter la simulation, il suffit de cliquer sur `stop`.

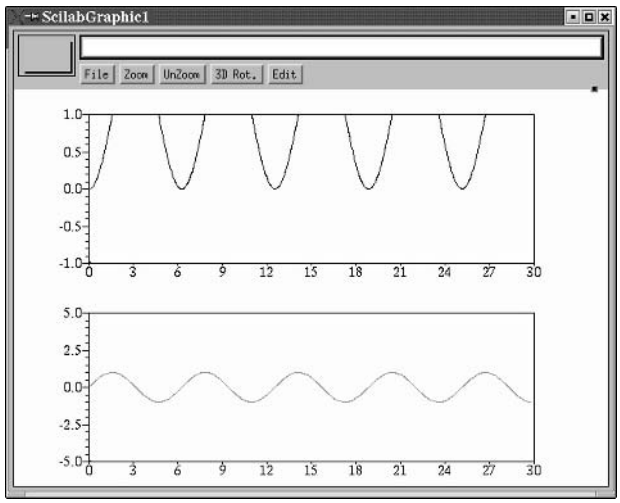


Figure 10.4 – Résultat de la simulation affiché par l'oscilloscope `MScope`.

10.1.4 Adapter les paramètres des blocs

On constate alors (voir figure 10.4) que les paramètres par défaut de l'oscilloscope ne sont pas bien réglés ; il faut faire en sorte que la première courbe reste dans la fenêtre graphique. Pour changer les paramètres d'un bloc, il suffit de cliquer sur le bloc, ce qui ouvre une fenêtre de dialogue (voir figure 10.5). On peut régler le problème en changeant les premières valeurs des vecteurs `Ymin_vector` et `Ymax_vector`, par exemple on peut remplacer la première (qui vaut -1) par 0 et la deuxième par 2.

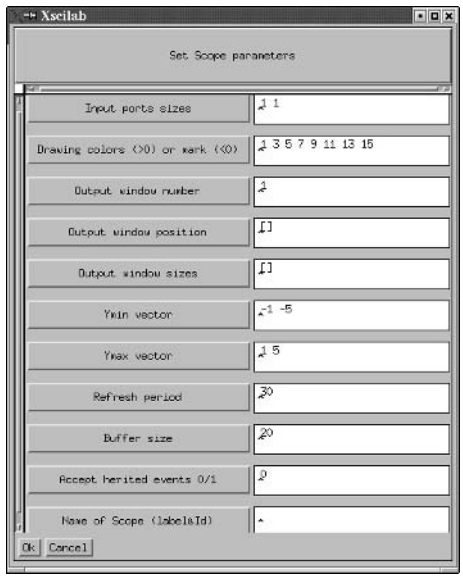


Figure 10.5 – Paramètres par défaut de l'oscilloscope.

10.1.5 Sauvegarde et rechargement

Un schéma peut, à tout moment, être sauvé dans un fichier grace au menu `Diagram/Save As` qui demande à l'utilisateur de spécifier un nom de fichier qui doit avoir l'extension `.cos` ou `.cosf`. L'extension `.cos` indique une sauvegarde en format binaire alors que `.cosf` indique une sauvegarde textuelle. Il est aussi possible de faire une sauvegarde rapide en utilisant le menu `Diagram/Save`. Dans ce cas le schéma est sauvé en format binaire dans le fichier `<nom du schéma>.cos`.

Les fichiers de sauvegarde peuvent être rechargés par le menu `Diagram/Load`.

10.2 Requins et sardines

On est maintenant en mesure de construire des modèles plus compliqués. Prenons par exemple le modèle de la population des requins et des sardines présenté dans le chapitre 9 :

$$\begin{cases} x'(t) = a\,x(t) - b\,x(t)y(t) & a, b > 0 \\ y'(t) = c\,x(t)y(t) - d\,y(t) & c, d > 0 \end{cases}$$

où $x(t)$ représente le nombre de sardines et $y(t)$ représente le nombre de requins. Ce modèle approché, appelé aussi système de Lotka-Volterra, signifie qu'en l'absence de requins les sardines prolifèrent $x'(t) = a\,x(t)$, qu'en l'absence de sardines les requins disparaissent $y'(t) = -d\,y(t)$; le terme en $x(t)y(t)$, qui

représente la rencontre des requins et des sardines, augmente le nombre de requins et diminue le nombre de sardines (car ces dernières sont mangées par les requins).

Le schéma Scicos représentant ce modèle est donné dans la figure 10.6 qui correspond au fichier source *scicos_ diag/scicos6000.cos*. On retrouve les états x et y comme les sorties des deux intégrateurs (un intégrateur pour chaque équation différentielle). Les entrées de ces intégrateurs sont alors x' et y'. Pour réaliser les équations définissant x' et y' dans le schéma, on a utilisé des blocs sommateurs, multiplicateurs et gains (multiplication par constante). On a pris $a = 2$, $b = 1$, $c = 0,3$ et $d = 1$. Le bloc `Scopexy` est utilisé pour visualiser le

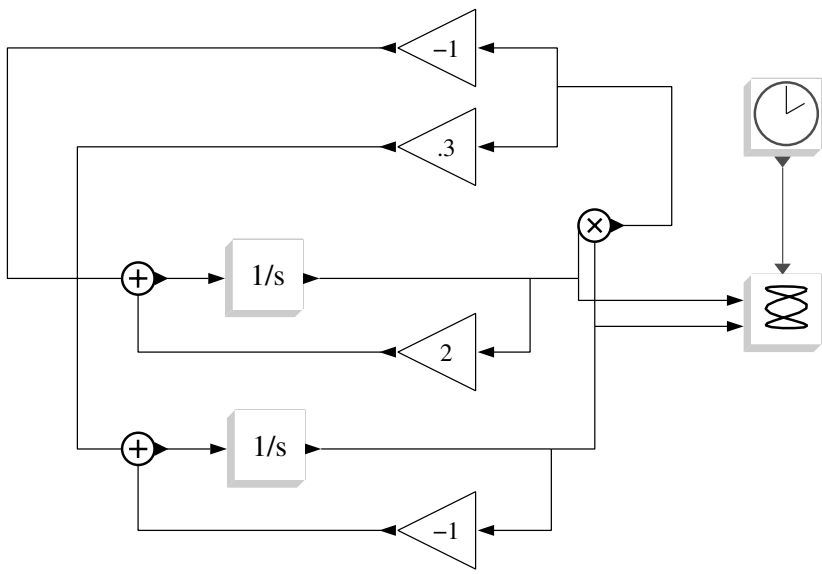


Figure 10.6 – Le schéma Scicos correspondant au modèle de la population des requins et des sardines.

résultat. Ce bloc affiche la deuxième entrée en fonction de la première.

Le résultat de la simulation pour $x(0) = 2$ et $y(0) = 1$ est donné par la figure 10.7. On retrouve le comportement périodique attendu. On constate qu'il peut exister une forte variation de la population des sardines ; cela dépend des populations initiales des sardines et des requins ($x(0)$ et $y(0)$). Si l'on commence près des populations d'équilibre, les variations sont moins fortes. Les populations d'équilibre x_e et y_e correspondent à une condition initiale pour

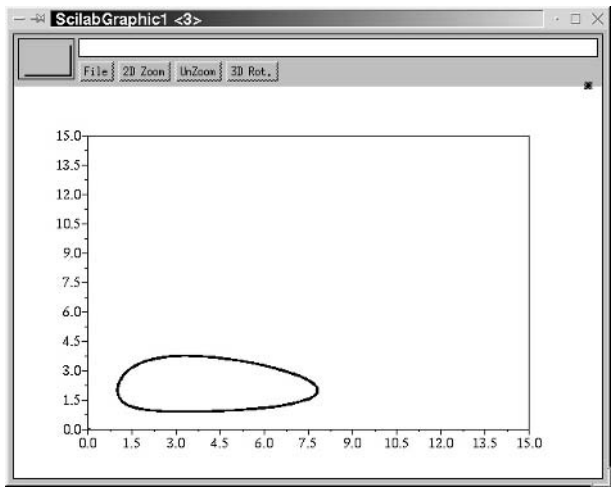


Figure 10.7 – L'évolution de la population des requins en fonction de celle des sardines.

laquelle l'état reste constant. Elles vérifient bien entendu le système :

$$\begin{cases} 0 = a\,x_e - b\,x_e y_e \\ 0 = c\,x_e y_e - d\,y_e \end{cases}$$

ce qui (à part la solution évidente $(0,0)$) donne $x_e = d/c$ et $y_e = a/b$.

10.2.1 Régulation par la pêche

On peut introduire l'influence de la pêche des sardines dans le modèle précédent en modifiant l'équation de $x'(t)$ comme suit :

$$x'(t) = a\,x(t) - b\,x(t)y(t) + f\,x(t) \qquad a, b > 0, \quad f < 0,$$

où le coefficient f indique l'effort de pêche. Noter que cette modification ne modifie pas le point d'équilibre x_e.

Supposons maintenant que les autorités de pêche veuillent réguler la population des sardines. Pour cela ils autorisent la pêche ($f < 0$) si la population des sardines dépasse un certain seuil ($x > x_{\max}$) et ils interdisent la pêche ($f = 0$) quand cette population passe en dessous d'un autre seuil ($x < x_{\min}$). Pour modéliser ce régulateur, on note qu'il faut générer deux événements : ces événements sont alors utilisés pour modifier la valeur de f (voir figure 10.8 qui correspond au fichier source *scicos_diag/Predator_prey_fishing000.cos*). Les blocs `-to+` et `+to-` génèrent des événements quand leurs entrées traversent zéro dans le sens négatif vers positif et positif vers négatif, respectivement. Le bloc `Selector` copie la valeur (ici 0) qui se trouve sur sa première entrée sur sa sortie

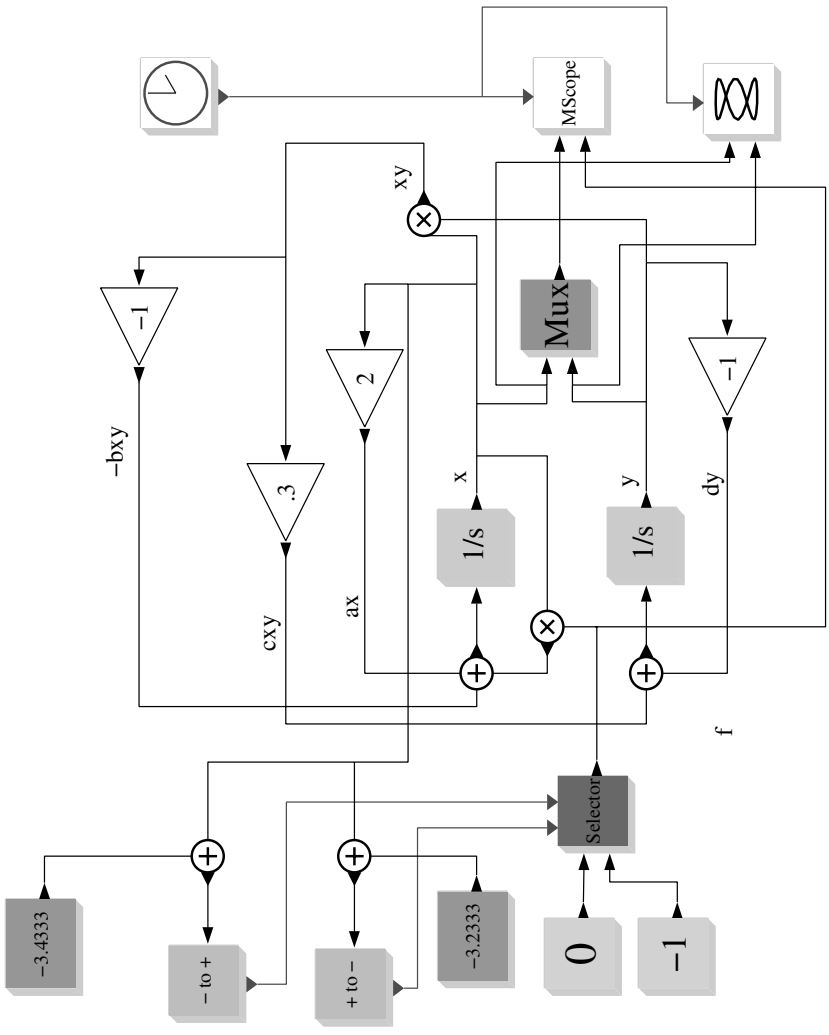


Figure 10.8 – Un modèle Scicos de l'évolution de la population des sardines et des requins régulée par la pêche.

quand il reçoit un événement sur son premier port d'entrée d'activation, et il copie la valeur sur sa deuxième entrée (ici 1) quand il reçoit un événement sur l'autre port. Donc quand $x - 3.43$ dépasse 0, f devient -1 et quand $x - 3.23$ passe en dessous de 0, f devient 0. Ces valeurs sont choisies pour faire tendre x vers sa valeur d'équilibre $x_e = d/c = 3.33$. Le bloc `Mux` permet de concaténer les deux entrées formant ainsi un signal vectoriel. Ce signal est affiché par `MScope` en superposant les deux valeurs (voir figure 10.9). La courbe inférieure indique les périodes d'ouverture et de fermeture de la pêche.

Les fonctions x et y sont aussi affichées par un bloc `Scopexy`. Notons qu'il faut modifier le paramètre `Ouput window number` pour éviter que les deux blocs d'affichage utilisent la même fenêtre graphique (voir la figure 10.10).

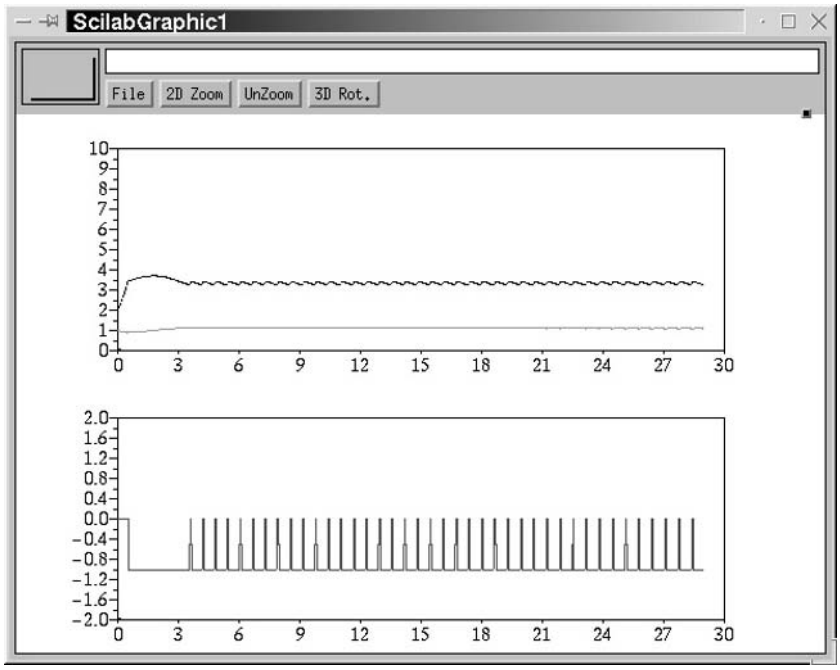


Figure 10.9 – L'évolution des populations dans le cas régulé. En haut, en noir, on a la population des sardines et, en gris, la population des requins. En bas, on trouve les périodes d'ouverture de la pêche ($f = -1$).

10.3 Super Bloc

Le schéma 10.8 est très chargé et donc on aimerait bien utiliser une description hiérarchique pour le simplifier et le rendre plus lisible. Pour cela on peut utiliser un « Super Bloc », qui ressemble à un bloc normal mais qui peut

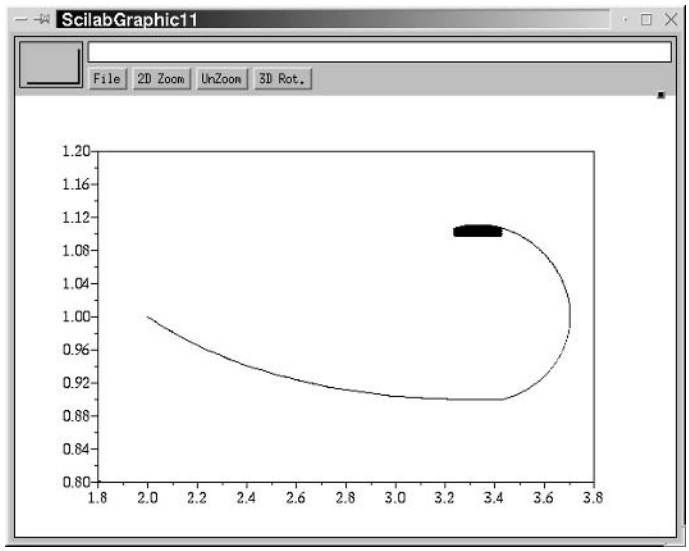


Figure 10.10 – La variation de la population des sardines en fonction de la population des requins.

contenir tout un sous-schéma.

Par exemple, pour mettre la partie régulateur dans un Super Bloc, on peut cliquer sur le bouton `Region_to_SuperBlock` du menu `Diagram`, puis sélectionner la région correspondant au régulateur. Cette région disparaît alors et elle est remplacée par un seul bloc de type « Super Bloc » (voir la figure 10.11 qui correspond au fichier source :*scicos_diag/Predator_prey_fishing2000.cos*). Pour accéder au schéma qui se trouve à l'intérieur de ce bloc, il suffit de cliquer dessus, ce qui lance un nouvel éditeur Scicos.

On peut manipuler un « Super Bloc » comme n'importe quel autre bloc. On peut, en particulier, le copier et l'utiliser plusieurs fois dans le même schéma. De manière générale, on peut utiliser un nombre illimité de « Super Bloc » par schéma et on peut placer des « Super Bloc » dans d'autres « Super Bloc ».

10.4 Paramètres formels

Il est possible, et souvent souhaitable, de définir les paramètres des blocs Scicos de manière formelle. Dans l'exemple précédent, au lieu d'écrire dans les blocs les valeurs numériques des paramètres a, b, c, etc., on peut les définir dans le « Context » du schéma comme variables Scilab puis utiliser leur noms comme paramètres de blocs. Le « Context » peut être invoqué en cliquant sur le bouton `Edit/Context` (voir figure 10.12). On peut mettre n'importe quelle instruction Scilab dans le « Context », le contenu étant exé-

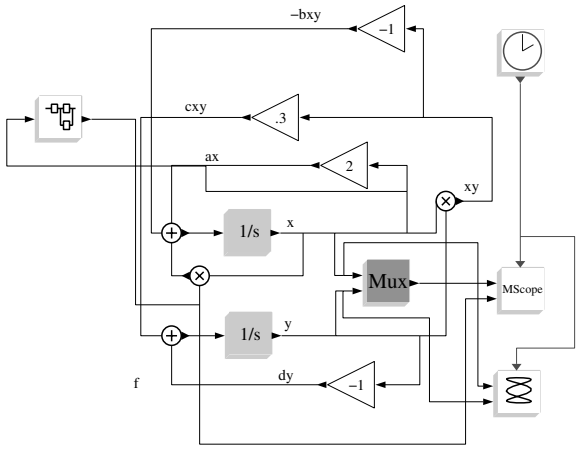


Figure 10.11 – Mise en Super Bloc du régulateur.

cuté comme un script. Une fois ces variables définies, on peut les utiliser

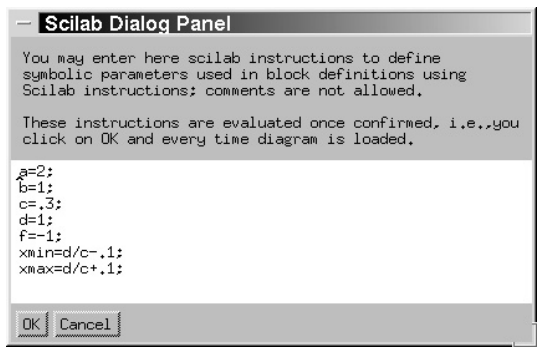


Figure 10.12 – Le « Context » du schéma Scicos.

comme paramètres de blocs (figure 10.13 qui correspond au fichier source :*scicos_diag/Predator_prey_fishing_symbolic2000.cos*).

Chaque Super Bloc a un « Context » qui peut être utilisé pour définir des paramètres locaux au Super Bloc. De plus, un paramètre défini dans le « Context » d'un Super Bloc est aussi utilisable dans les blocs de tout Super Bloc contenu dans ce Super Bloc. Autrement dit, quand Scicos rencontre un paramètre formel dans un bloc, il cherche le « Context » du Super Bloc contenant le bloc pour sa définition. S'il ne la trouve pas, il cherche dans le « Context »

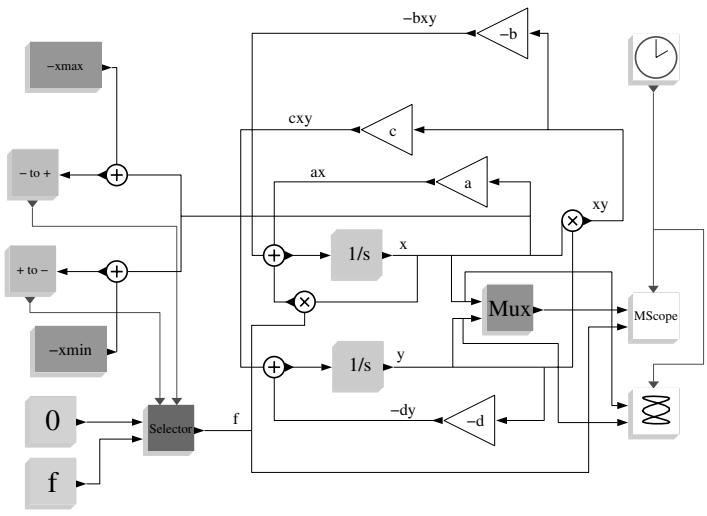


Figure 10.13 – Le modèle régulé avec le « Context » défini dans la figure 10.12 (à comparer avec la figure 10.8).

du Super Bloc contenant le Super Bloc, et ainsi de suite jusqu'au « Context » du diagramme principal.

Le « Context » est sauvegardé automatiquement avec le schéma et il est évalué à chaque chargement. Il est aussi réévalué après chaque modification.

10.5 Construction de nouveaux blocs

Les blocs pré-définis dans les palettes standard de Scicos permettent de construire des schémas très divers mais, dans certains cas, une fonctionnalité décrite à partir de plusieurs blocs peut être réalisée par un seul bloc, ce qui donne souvent une simulation plus efficace. On a déjà vu une façon de remplacer plusieurs blocs par un seul en utilisant le Super Bloc. Cela permet de construire des schémas hiérarchiques plus lisibles mais n'améliore pas l'efficacité de la simulation car la première étape de la compilation dans Scicos consiste à mettre à plat le diagramme. En fait, un Super Bloc n'est qu'une facilité graphique pour masquer des sous-diagrammes ; ce n'est pas un bloc de base, même s'il a été converti en un bloc par le menu `Diagram/Save_as_Interf_func`.

Un nouveau bloc de base peut être défini de plusieurs façons, mais dans tous les cas on a besoin de deux fonctions :

- une **fonction d'interface**, presque toujours écrite en langage Scilab, pour gérer l'interface avec l'éditeur Scicos et avec l'utilisateur
- une **fonction de simulation** réalisant le comportement dynamique du bloc. Cette fonction est utilisée par le simulateur et doit donc être rapide. Elle est souvent en C, mais elle peut aussi être en langage Scilab, en particulier dans la phase de mise au point.

Les codes source de toutes les fonctions d'interface et de simulation de tous les blocs des palettes de Scicos sont disponibles et peuvent être consultés à travers les pages d'aides des blocs. Il faut noter que les blocs `Event select` et `If-Then-Else` utilisés pour le conditionnement d'événements ne sont pas des blocs réguliers, et en particulier, ils n'ont pas de fonction de simulation.

10.5.1 Fonction d'interface

La fonction d'interface d'un bloc détermine non seulement sa géométrie, sa couleur de fond, son icône, le nombre de ses ports d'entrée-sortie et leurs tailles respectives etc., mais aussi ses états initiaux et ses paramètres. De plus, elle gère le dialogue qui permet de modifier ses propriétés, ce que l'utilisateur peut faire en cliquant sur le bloc.

Si le bloc que l'on veut définir n'a pas un aspect très particulier (c'est-à-dire s'il est rectangulaire, avec des ports placés aux endroits habituels et une icône ne contenant qu'un texte), il est assez facile de construire sa fonction d'interface en copiant celle d'un bloc pré-défini. La seule partie qu'il faut adapter est la gestion du dialogue avec l'utilisateur, permettant de modifier les paramètres du bloc.

10.5.2 Fonction de simulation

Cette fonction est appelée au cours de la simulation pour effectuer principalement les tâches suivantes :

- **Initialisation :** le simulateur appelle cette fonction une fois tout au début de la simulation pour lui permettre d'initialiser ses états initiaux (si nécessaire car les états sont aussi initialisés par la fonction d'interface). À cette occasion, la fonction peut effectuer d'autres tâches comme par exemple ouvrir et initialiser une fenêtre graphique ou ouvrir un fichier. À ce stade les entrées du bloc ne sont pas disponibles.
- **Ré-initialisation :** la fonction est appelée plusieurs fois juste après l'initialisation pour permettre à chaque bloc d'imposer des contraintes sur ses entrées-sorties et son état.
- **Calcul des sorties :** la fonction calcule ses sorties en fonction des valeurs de ses entrées et de ses états.
- **Mise à jour des états :** la fonction met à jour son état discret en fonction de ses états courants et de ses entrées. Elle peut faire de même

avec son état continu (pour modéliser des sauts éventuels).
- **Calcul de la dérivée de l'état continu :** si le bloc contient un état continu, la fonction de simulation calcule sa dérivée.
- **Calcul des dates des événements de sortie :** si le bloc possède des ports de sortie d'activation, la fonction fournit l'information concernant les dates des prochaines générations de ces événements.
- **Terminaison :** tous les blocs sont appelés une fois à la fin de la simulation pour effectuer, si nécessaire, des tâches telles que la fermeture des fichiers, etc.

Dans certains cas, la fonction de simulation peut aussi être appelée pour effectuer d'autres tâches comme le calcul des modes du bloc. L'utilisation de ces fonctionnalités nécessite une bonne connaissance du formalisme Scicos et ne sera pas présentée dans cet ouvrage. L'utilisateur a rarement besoin de construire des nouveaux blocs nécessitant ces fonctionnalités.

La fonction de simulation peut être en C ou en Scilab. Une fonction en C serait comme suit :

```
#include "scicos_block.h"
#include <math.h>

void  my_block(scicos_block *block,int flag)
{
  ...
}
```

Le drapeau (« flag ») dans la liste d'appel indique la tâche à effectuer :

Drapeau	Tâche
0	Calcul de la dérivée de l'état continu
1	Calcul des sorties
2	Mise à jour des états
3	Calcul des dates des événements de sortie
4	Initialisation
5	Terminaison
6	Ré-initialisation
9	Calcul des fonctions de traversée de zéro et des modes

L'autre argument de la liste d'appel de la fonction de simulation est une structure qui contient les états, les entrées, les sorties, les paramètres, etc., et leurs dimensions. On y trouve aussi un entier (indicateur d'activation) renseignant sur la manière dont la fonction a été appelée (le bloc correspondant a été activé). La structure `scicos\_block` est définie comme suit :

```
typedef struct {
  int nevprt;     /* indicateur d'activation, codage binaire */
  voidg funpt;    /* pointeur vers la fonction de simulation */
  int type;       /* type de fonction de simulation, (4 ou 5) */
```

```
  int scsptr;     /* pas utilisé pour les fonctions C */
  int nz;         /* taille de z */
  double *z;      /* vecteur d'état discret */
  int nx;         /* taille de x */
  double *x;      /* vecteur d'état continu */
  double *xd;     /* dérivée de l'état continu */
  double *res;    /* pas utilisé pour les blocs explicits */
  int nin;        /* nombre d'entrées */
  int *insz;      /* tailles des entrées */
  double **inptr; /* table de pointeurs vers les entrées */
  int nout;       /* nombre de sorties */
  int *outsz;     /* tailles des sorties */
  double **outptr;/* table de pointeurs vers les sorties */
  int nevout;     /* nombre de sorties d'activation */
  double *evout;  /* retard des événements de sorties */
  int nrpar;      /* taille de  rpar */
  double *rpar;   /* vecteur des paramètres réels */
  int nipar;      /* taille de ipar */
  int *ipar;      /* vecteur des paramètres entiers */
  int ng;         /* nombre de zero-crossings */
  double *g;      /* les surfaces de zero-crossing */
  int ztyp;       /* 1 si le bloc contient un zero-crossing */
  int *jroot;     /* direction des zero-crossings */
  char *label;    /* label du bloc */
  void **work;    /* la mémoire allouée par le bloc*/
  int nmode;      /* nombre de modes, taille de mode */
  int *mode;      /* vecteur de modes */
} scicos_block;
```

La fonction de simulation peut avoir accès à d'autres informations à travers les fonctions :

- **double get_scicos_time()** qui retourne le temps `t` ;
- **int get_phase_simulation()** qui retourne la phase de la simulation (1 pour une activation par événement et 2 pour la phase d'intégration numérique) ;
- **int get_block_number()** qui retourne le numéro du bloc.

La fonction de simulation en C à deux arguments est de type 4. La version Scilab de cette fonction est de type 5 et elle a la liste d'appel suivante :

```
block=func_name(block,flag)
```

où `flag` est un nombre et `block` une structure Scilab (`tlist`) ayant les champs suivants :

```
nevprt funpt  type  scsptr nz    z    nx     x
xd     res    nin   insz   inptr nout outsz  outptr
nevout evout  nrpar  rpar  nipar ipar ng     g
ztyp   jroot  label  work  nmode mode
```

Les champs correspondent aux champs de la structure C **scicos_block** utilisé dans la construction des fonctions de type 4.

Les fonctions Scilab suivantes qui peuvent être utilisées à l'intérieur d'une fonction de simulation de type 5 sont les suivants :

- **curblock()** qui retourne le numéro de bloc ;
- **scicos_time()** qui retourne le temps ;
- **phase_simulation()** qui retourne la phase ;
- **set_blockerror(i)** qui permet de signaler une erreur au simulateur.

Actuellement, tous les nouveaux blocs doivent être écrits en type 4 ou 5. Mais d'autres types ont été utilisés dans le passé et continuent à fonctionner dans Scicos.

La fonction de simulation est appelée quand le bloc correspondant est activé. Après les initialisations et au cours de la simulation, cela se produit parce que le bloc reçoit un événement sur l'un de ses ports d'entrée d'activation ; dans le cas d'absence de ce type de port, il hérite d'un événement par l'une ou plusieurs de ses entrées. L'indicateur d'activation indique par quelle voie le bloc a été activé (1 si l'événement d'activation a été reçu sur le premier port d'entrée d'activation, 2 si c'est sur le deuxième, 3 si c'est simultanément sur le premier et le deuxième, 4 si c'est sur le troisième, etc.). Le bloc peut aussi être activé sans avoir reçu d'événement à condition qu'il soit déclaré comme étant dépendant du temps, c'est-à-dire actif en permanence. Dans ce cas, l'indicateur d'activation vaut 0.

À chaque activation du bloc en cours de simulation, la fonction de simulation peut être appelée une ou plusieurs fois. Dans le cas d'un simple bloc réalisant une fonction immédiate, par exemple le bloc `Trig.Function`, la fonction n'est appelée qu'avec le drapeau 1. Un bloc contenant un état discret sera aussi appelé avec le drapeau 2, et si de plus le bloc a des ports de sorties d'activation, il sera appelé aussi avec le drapeau 3. L'ordre des appels, dans le cas d'une activation par événement, est le drapeau 1 suivi par le drapeau 3, et enfin le drapeau 2. Dans le cas d'activation permanente, c'est 1 puis 0.

Exemple On va reprendre ici l'exemple du système dynamique modélisant la régulation des populations de requins et de sardines du schéma de la figure 10.13. Dans ce schéma, presque la moitié des blocs sont utilisés pour implanter le modèle de la population des requins et des sardines, c'est-à-dire, à un changement de notation près :

$$\begin{cases} x_0'(t) = a\,x_0(t) - b\,x_0(t)x_1(t) + f\,x_0(t), & a, b > 0, \quad f < 0, \\ x_1'(t) = c\,x_0(t)x_1(t) - d\,x_1(t), & c, d > 0 \end{cases}$$

Ici $x_0(t)$ représente le nombre de sardines et $x_1(t)$ représente le nombre de requins.

Les autres blocs sont dédiés à la partie régulation et à la visualisation. Il serait donc intéressant de définir avec un seul bloc le modèle de la population. Ce bloc aurait comme entrée f (l'effort de pêche) et le vecteur `x=[x0,x1]` pour

l'état continu et la sortie. Les paramètres seraient `rpar=[a,b,c,d]`. Il remplacerait alors tous les blocs dédiés au modèle de la population et simplifierait ainsi le schéma de simulation.

La fonction de simulation de ce bloc pourrait être écrite (en C) comme suit :

```
#include <scicos/scicos_block.h>
#include <math.h>
void pred_prey(scicos_block *block,int flag)
{
  if (flag == 1 || flag==6) {
    /*  y=x  */
    block->outptr[0][0]=block->x[0];
    block->outptr[1][0]=block->x[1];
  } else if (flag == 0) {
    /*     x0d = a*x0-b*x0*x1+f*x0 */
    /*     x1d = c*x0*x1-d*x1 */
    block->xd[0]=block->rpar[0]*block->x[0]-block->rpar[1]*
      block->x[0]* block->x[1]+block->inptr[0][0]*block->x[0];
    block->xd[1]=block->rpar[2]*block->x[0]*
      block->x[1]-block->rpar[3]*block->x[1];
  }
return;
}
```

Fichier source : `scicos_diag/pred_prey.c`

Ce bloc ne sera appelé qu'avec les drapeaux 0, 1, 4, 5 et 6. Pour les drapeaux 4 et 5, il n'y a clairement rien à faire. Pour les drapeaux 1 et 6, il faut calculer les sorties, c'est-à-dire recopier l'état dans les sorties. Enfin, pour le drapeau 0, il faut calculer la dérivée de x que l'on place dans `block->xd` (c'est-à-dire x').

Pour utiliser ce bloc dans Scicos, il faut compiler et linker `pred_prey` dans Scilab. Si la fonction `pred_prey` se trouve dans le fichier `pred_prey.c`, cela peut se faire comme suit (voir le paragraphe 7.4 pour plus de détails) :

```
->ilib_for_link("pred_prey","pred_prey.o",[],"C")

->exec loader.sce
```

La fonction de simulation `pred_prey` est maintenant utilisable dans Scicos pour la construction du bloc `Pred-Prey`. Il ne nous reste donc qu'à écrire la fonction d'interface.

Dans un premier temps, on utilise la fonction d'interface canonique `GENERIC`. On commence par supprimer tous les blocs que l'on veut remplacer dans le diagramme. Puis on copie le bloc `GENERIC` qui se trouve dans la palette `Others` et on le place dans le diagramme. En cliquant dessus, on ouvre une fenêtre de dialogue qui permet de choisir ses paramètres (voir les figures 10.14 et 10.15).

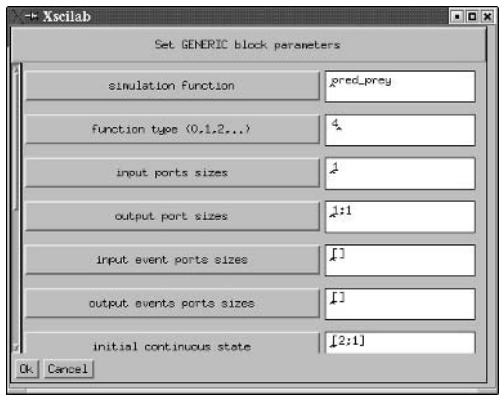


Figure 10.14 – Le dialogue du bloc `GENERIC` : le nom de la fonction de simulation, son type, les tailles et les nombres d'entrées-sorties, etc.

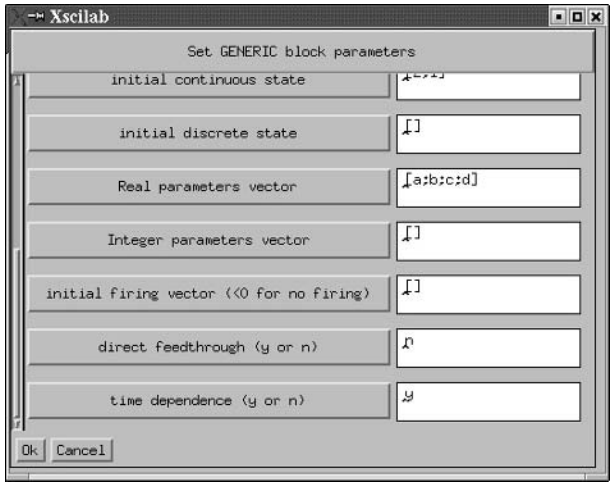


Figure 10.15 – Le dialogue du bloc `GENERIC` (suite).

Remarquer les deux dernières questions. La première concerne l'existence d'une dépendance directe entrée-sortie. Dans ce cas, les sorties x_0 et x_1 ne dépendent pas directement de l'entrée f (seule la dérivée de x_0 en dépend, on considère cela comme une dépendance indirecte). La deuxième question concerne la propriété de dépendance en temps ou plus précisément l'activation permanente. La réponse est affirmative car ce bloc doit être activé en permanence, il n'a pas besoin des signaux d'activation.

Après avoir cliqué sur `OK`, on constate que le bloc `GENERIC` se redessine avec un port d'entrée et deux ports de sortie. On peut alors brancher les entrées et les sorties pour obtenir le schéma de la figure 10.16 qui correspond au fichier source : *scicos_diag/Predator_prey_mod2000.cos*.

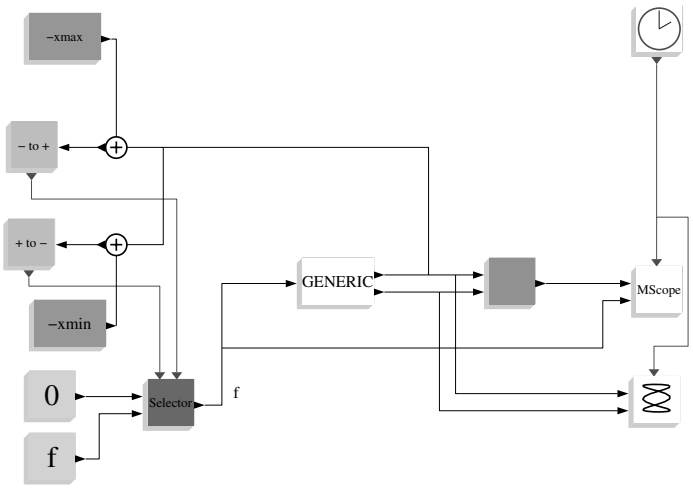


Figure 10.16 – L'utilisation du bloc `GENERIC`.

La simulation du nouveau schéma montre qu'il est identique au précédent en fonctionnalité et un peu plus rapide en ce qui concerne la vitesse de simulation.

Ce bloc peut maintenant être copié, placé dans une palette, etc., comme n'importe quel autre bloc Scicos. Mais l'utilisation de ce bloc n'est pas conviviale. On peut bien entendu adapter le bloc en changeant sa couleur et son icône, mais quand on clique dessus, on a toujours le même dialogue canonique dont la moitié des questions ne sont pas pertinentes.

Il faut donc construire une fonction d'interface pour présenter un dialogue spécifique avec éventuellement des tests de validité (par exemple ne pas accepter des paramètres a, b, c, d négatifs). Une fonction d'interface qui permet de faire cela est la suivante :

```
function [x,y,typ]=P_Prey(job,arg1,arg2)
x=[];y=[];typ=[]
select job
case "plot" then
  standard_draw(arg1)
case "getinputs" then
  [x,y,typ]=standard_inputs(arg1)
case "getoutputs" then
  [x,y,typ]=standard_outputs(arg1)
case "getorigin" then
  [x,y]=standard_origin(arg1)
case "set" then
  x=arg1
  graphics=arg1.graphics;exprs=graphics.exprs
  model=arg1.model;
  while %t do
    [ok,a,b,c,d,x0,exprs]=getvalue(..
        "Définissez les paramètres",..
["a";"b";"c";"d";"Etat initial"],..
list("vec",1,"vec",1,"vec",1,"vec",1,"vec",2),exprs)
    if ~ok then break,end
    if a<0 | b<0 | c<0 | d<0 then
      message("les paramètres doivent être positifs")
    else
      if ok then
        graphics.exprs=exprs;x.graphics=graphics;
        model.state=x0(:);model.rpar=[a;b;c;d];x.model=model
        break
      end
    end
  end
case "define" then
  x0=[2;1];a=2;b=1;c=.3;d=1;  //default
  model=scicos_model()
  model.sim=list("pred_prey",4)
  model.in=1;model.out=[1;1]
  model.state=x0;
  model.rpar=[a;b;c;d]
  model.dep_ut=[%f %t]
  exprs=[string(a);string(b);string(c);
         string(d);strcat(sci2exp(x0))]
  gr_i="xstringb(orig(1),orig(2),""Pred_prey"",sz(1),sz(2),..
                 ""fill"")"
  x=standard_define([3 2],model,exprs,gr_i)
end
endfunction
```

Fichier source : `scicos_diag/P_Prey.sci`

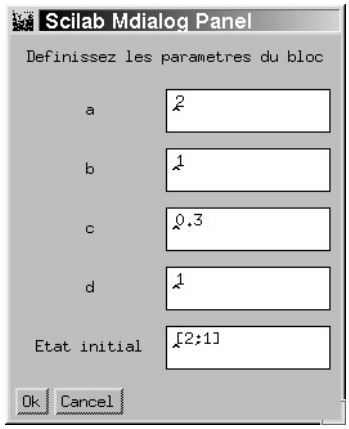
On a utilisé ici les fonctions standards pour les cas `plot` (représentation du bloc), `getinputs` (gestion des coordonnées des entrées), `getoutputs` (gestion des coordonnées des sorties), `getorigin` (gestion des coordonnées du bloc dans le schéma). Cela peut se faire pour tous les blocs de forme rectangulaire. Les cas `set` et `define` sont par contre spécifiques. Le premier gère le dialogue avec l'utilisateur et le deuxième définit les valeurs initiales des paramètres.

Pour comprendre cette fonction, il faut savoir qu'un bloc Scicos est défini par une structure Scilab contenant les champs `graphics`, `model`, `gui`, `doc`. Les champs `graphics` et `model` contiennent aussi des structures.

La structure graphique `graphics` contient des informations sur l'aspect graphique du bloc comme sa taille, son emplacement, son orientation, etc., et `model` des informations nécessaires pour la simulation comme le nom de la fonction de simulation et son type, les nombres et les tailles des ports d'entrée-sortie, les valeurs des états, des paramètres, etc. Enfin, `gui` est une chaîne de caractères Scilab contenant le nom de la fonction d'interface associée à ce bloc.

La variable `model` est initialisée dans le cas `define`. On constate alors que le nom de la fonction de simulation est `pred_prey`, qui est de type 4, que le bloc a une seule entrée de taille 1 et deux sorties de taille 1 chacune, que son état continu initial est $[2; 1]$ et qu'il a comme paramètre réel le vecteur $[2; 1; 0.3; 1]$. Remarquer aussi l'élément `[%f %t]` du `model` de notre exemple, qui indique que ce bloc ne contient pas de dépendance directe entrée-sortie, mais qu'il est activé en permanence. Le champ `doc` est utilisé pour la documentation.

Pour utiliser ce bloc, il suffit maintenant d'utiliser le bouton `AddNewBlock` du menu `Edit`. On peut alors remplacer le bloc `GENERIC` par le bloc `Pred_prey`. La simulation donne alors exactement le même résultat qu'avant, sauf que maintenant en cliquant sur le bloc on obtient :



Pour construire des blocs sophistiqués, on peut chercher un bloc pré-défini ayant des caractéristiques semblables à ceux que l'on cherche à construire pour en examiner les codes source. Pour examiner les fonctions associées à un bloc, voir sa page "help".

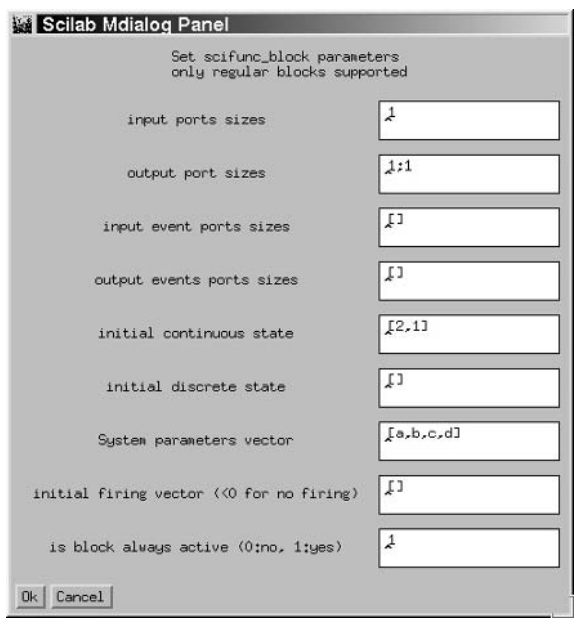
10.6 Blocs génériques

Outre le bloc `GENERIC` dont on a vu précédemment l'utilisation la palette `Others` contient deux blocs génériques `Scifunc` et `Cblock2` facilitant la définition de nouvelles fonctions de simulation.

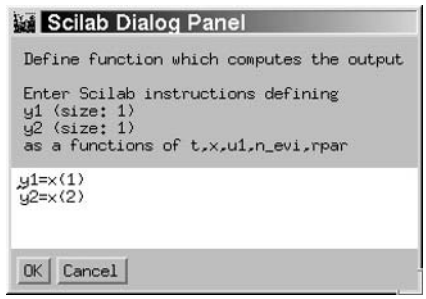
10.6.1 Bloc Scifunc

Le bloc `Scifunc` permet le prototypage rapide des nouveaux blocs dont la fonction de simulation est définie par du code Scilab. Ce bloc permet de construire et de tester des blocs réguliers généraux. Outre une fonction d'interface générique, ce bloc propose une édition en ligne assistée de la fonction de simulation, en langage Scilab.

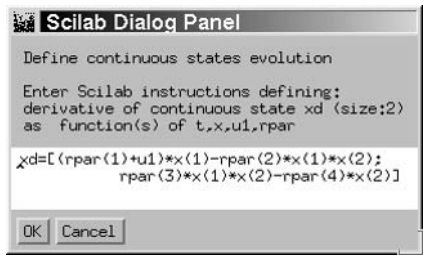
Pour présenter le fonctionnement de ce bloc, on reprend le schéma de la figure 10.16 et on remplace le bloc `GENERIC` par un `Scifunc`. On commence par supprimer le bloc `GENERIC`, puis on copie le bloc `Scifunc` de la palette `Others` à sa place. En cliquant sur celui-ci, on a le dialogue suivant :



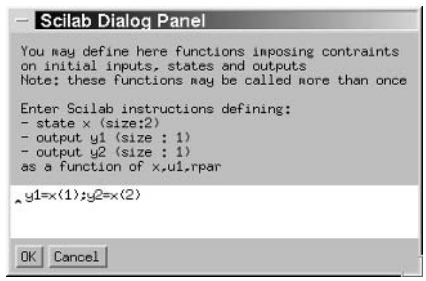
En cliquant sur `OK`, on lance l'édition de la fonction de simulation. On commence par définir la sortie :



puis la dérivée de l'état continu :



et enfin la phase d'initialisation :



En cliquant sur `OK`, `Scifunc` se redessine avec le bon nombre de ports d'entrée et de sortie, et il suffit alors de les brancher pour que le schéma devienne simulable. Bien entendu, le résultat de la simulation est identique au cas précédent, mais la vitesse de simulation est réduite.

Le bloc `Scifunc` doit être utilisé seulement dans la phase de mise au point des nouveaux blocs sauf peut-être pour des blocs qui ne sont pas activés souvent pendant la simulation. Dans l'exemple présenté ici, `Scifunc` contient un modèle dynamique continu, il est donc appelé très souvent par le solveur de Scicos. C'est pour cela que son utilisation dans ce cas est particulièrement coûteuse en termes de rapidité de la simulation.

10.6.2 Bloc `Cblock2`

Le bloc `Cblock2` propose aussi une fonction d'interface générique. Dans ce bloc, la fonction de simulation est définie textuellement en C dans la phase de

dialogue avec la fonction d'interface. L'intérêt est que la fonction de simulation fait partie intégrante du schéma Scicos contenant le bloc ; ainsi, en sauvegardant le schéma sous format `cos` ou `cosf`, on sauvegarde aussi la fonction de simulation sous forme textuelle. En ouvrant le fichier contenant un tel schéma dans Scicos, même sur une machine différente de celle sur laquelle le schéma a été créé, on obtient un schéma complet et simulable car les définitions textuelles des fonctions de simulation des blocs `Cblock2` sont copiées dans des fichiers, dans un répertoire temporaire, puis compilées et liées de manière transparente. Bien entendu, il faut que la machine soit équipée des compilateurs nécessaires.

10.7 Exemple

À travers un exemple simple, nous considérons le type de modèles que le formalisme Scicos permet de construire. Cet exemple ne correspond pas à un vrai cas d'application car le modèle est très simplifié.

Notre objectif ici est de construire un modèle de suspension et de simuler son comportement. Pour cela nous considérons que la suspension est montée sur un véhicule qui avance avec une vitesse constante sur une route qui n'est pas parfaitement lisse. Les inégalités de la route sont modélisées par des rectangles de taille et de placement aléatoires.

Le modèle de la suspension utilisé est un système linéaire d'ordre 2 :

$$y''(t) + ay'(t) + k(y(t) - z(t)) = 0$$

où y représente l'écart du véhicule par rapport à son assiette, z le profil de la route (égal à zéro en absence d'inégalité) et a et k sont les paramètres de la suspension.

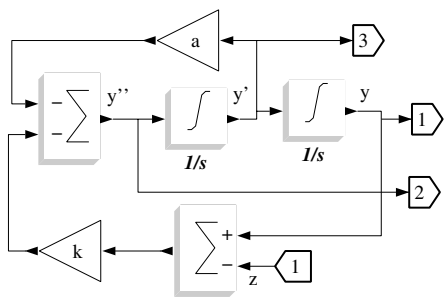


Figure 10.17 – Le modèle de suspension.

La figure 10.17 (qui correspond au fichier source :*scicos_ diag/susp3.cos*) présente le schéma Scicos du modèle de la suspension. Ce modèle est réalisé avec deux blocs intégrateurs à l'intérieur d'un super bloc. Cela est illustré dans

la figure 10.17. Ce super bloc, nommé “Suspension Model” reçoit en entrée z est génère y, y' et y''.

Le profile de la route $z(t)$ est une fonction constante par morceaux. Il vaut zéro sauf pendant des intervalles de temps de longueur aléatoire où il prend une valeur aléatoire positive. Pour modéliser $z(t)$, on génère deux types d'événements, le premier indique le début de chaque intervalle et le deuxième, la fin. Le super bloc de la figure 10.18 génère ces événements et la variable aléatoire positive qui représente la valeur de z à l'intérieur de l'intervalle. Le bloc `Random Generator` produit à chaque activation trois nombres aléatoires indépendants suivant une loi uniforme sur $[0, 1]$. Les deux premiers passent par des blocs `Log` et se multiplient par `-la` et `-mu` respectivement. Ils deviennent donc des variables aléatoires de loi exponentielle de paramètres `la` et `mu` qui alimentent les blocs `Event Delay`[1]. Le bloc `Event Delay` génère un événement sur sa sortie avec un retard par rapport à son temps d'activation ; ce retard est donné par la valeur reçue sur son entrée régulière au moment d'activation. En bouclant la sortie du deuxième bloc `Event Delay` sur l'entrée du premier, on obtient deux suites d'événements qui seront utilisés pour préciser les instants de sauts dans $z(t)$. La dernière sortie du bloc `Random Generator` produit la hauteur du saut.

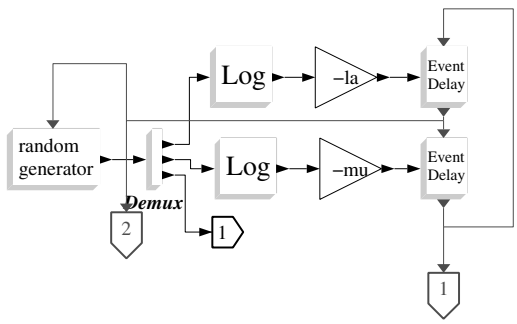


Figure 10.18 – La génération des variables aléatoires du modèle.

Le super bloc de la figure 10.18 (qui correspond au fichier source *scicos_ diag/susp1.cos*) est utilisé dans le super bloc nommé “Road Profile” de la figure 10.19 pour générer $z(t)$.

Les deux super blocs “Suspension Model” et “Road Profile” sont alors utilisés dans le schéma de la figure 10.20 (fichier source :*scicos_ diag/susp.cos*). Le « Context » de ce schéma contient les instructions suivantes :

```
k=1
a=.2
```

[1]Les blocs retards alimentés par des variables aléatoires de loi exponentielle sont souvent utilisés pour générer des processus de Poisson.

```
la=3
mu=.4
```

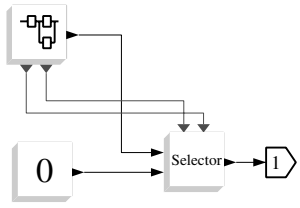


Figure 10.19 – La génération de $z(t)$.

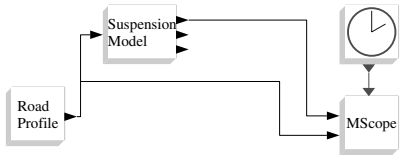
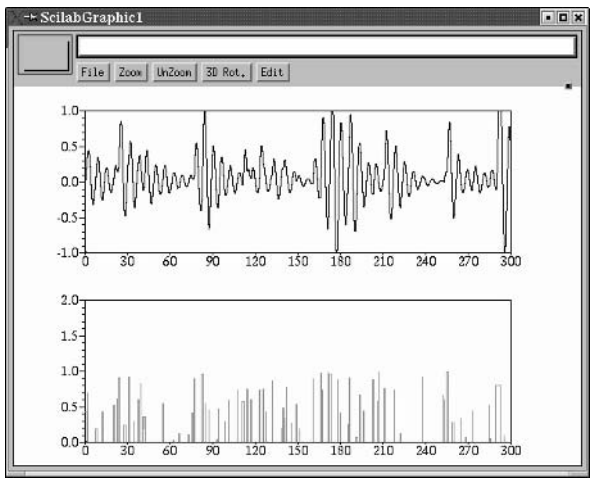


Figure 10.20 – Schéma complet.

La simulation donne le résultat suivant :



10.8 Traitement en mode batch

Dans certaines applications, il est souhaitable de pouvoir lancer des simulations Scicos à partir de Scilab sans passer par l'éditeur Scicos. Cela permet par exemple d'effectuer une série de simulations Scicos, en changeant un paramètre du modèle à chaque itération, et de stocker les résultats. On voit facilement l'intérêt de ce type de traitement dans les problèmes d'identification ou de réglage de paramètres de modèle.

Il existe deux fonctions dans Scilab pour lancer des simulations Scicos en mode batch : `scicosim` et `scicos_simulate`. Le premier est la fonction de base qui fait interface entre Scilab et Scicos, et son utilisation nécessite une connaissance des structures de données de Scicos. La fonction `scicos_simulate`, en revanche, est une fonction de haut niveau, basée sur `scicosim`, qui permet à l'utilisateur de programmer dans Scilab des changements de paramètres et des appels au simulateur Scicos. Cette fonction permet entre autre d'initialiser des variables avant l'exécution du « Context » du schéma Scicos. Cela permet de changer facilement les paramètres du modèle ; nous verrons comment dans l'exemple suivant.

Nous illustrons ici l'utilisation de la fonction `scicos_simulate` à travers un exemple. Nous considérons en particulier l'exemple de la suspension considéré dans la section précédente et le problème de réglage du paramètre a. Nous considérons deux critères pour le réglage de a. Le premier est une contrainte sur le pourcentage de temps où la valeur absolue de $y^{(3)}(t)$ dépasse un certain seuil. Le deuxième consiste à minimiser la fonction

$$\int_0^T (y(t)^2 + y'(t)^2)\, dt.$$

Pour calculer les valeurs correspondant à ces deux critères, nous complétons le schéma de la figure 10.20 en rajoutant les blocs nécessaires ; voir la figure 10.21 et le fichier source *scicos_ diag/suspension.cos*.

Pour permettre la modification du paramètre a, le « Context » du schéma est modifié comme suit :

```
k=1
if ~exists('a') then a=.2,end
la=3
mu=.4
```

Dans ce cas, la valeur de a est fixée à 0.2 sauf si elle a été définie auparavant.

Pour le calcul de $y^{(3)}$, nous utilisons un système linéaire ayant comme fonction de transfert $s/(1 + 0.01s)$ ce qui approche bien un dérivateur (nous n'utilisons pas le bloc dérivateur car y'' n'est pas dérivable). Le bloc `Mathematical Expression` contient l'expression `u1^2+u2^2` ce qui permet d'évaluer la fonction associée au critère numéro deux. La fonction du premier critère est évaluée en intégrant la sortie du bloc `Selector` qui génère dans ce cas un ou zéro en fonction des activations qu'il reçoit. Ces activations sont générées pas le bloc

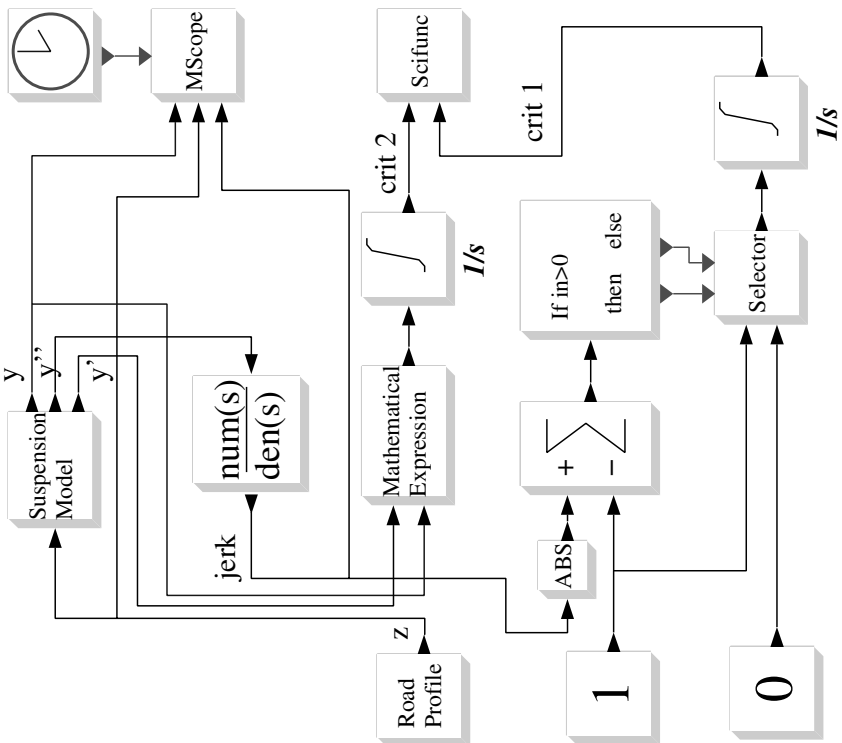


Figure 10.21 – Schéma complet avec l'évaluation des critères.

`If-Then-Else` qui en fonction du signe de son unique entrée régulière génère un signal d'activation sur l'un ou l'autre port de sortie.

Enfin, il reste le problème de renvoyer les résultats (les valeurs des deux fonctions associées aux deux critères) à Scilab. Il existe plusieurs méthodes dont les plus simples sont : passage par fichier et passage par variable globale. La méthode de passage par fichier consiste à utiliser un bloc d'écriture sur fichier pour sauvegarder le résultat dans un fichier et le relire dans Scilab. Cette méthode marche aussi dans l'autre sens, c.à.d. qu'elle permet de définir par exemple un signal dans Scilab et l'utiliser dans Scicos.

L'autre méthode, qui marche aussi dans les deux sens, consiste à échanger des données en utilisant des variables globales. Pour cela, il faut utiliser un bloc ayant comme fonction de simulation un programme Scilab, par exemple un `Scifunc` et qui aurait comme entrées les résultats à envoyer à Scilab. Il suffit alors de déclarer une variable globale et de copier les valeurs des entrées dans cette variable. C'est cette méthode que nous utilisons dans cet exemple.

Ici, nous n'avons besoin que des valeurs finales de `crit 1` et `crit 2`. Le bloc `Scifunc` ne sera donc pas activé. Dans ce cas, il est appelé seulement à l'initialisation et une fois à la fin. C'est dans le champ réservé aux instructions de la fin que nous mettons les commandes suivantes :

```
global Y
Y=[u1;u2]
```

ce qui mettra les valeurs désirées dans la variable globale `Y` accessible à partir de Scilab.

Le script Scilab ci-dessous permet d'effectuer des simulations pour des valeurs de a allant de 0.1 à 5, par pas de 0.1, et de faire le tracé des deux critères en fonction de a :

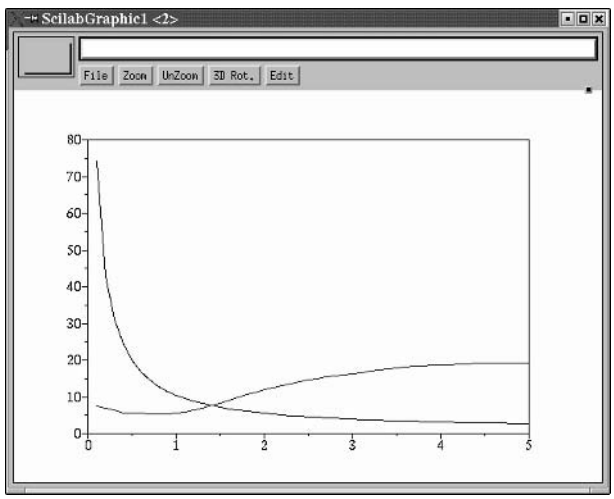
```
load suspension.cos  // chargement de schema (scs_m)
global Y   // definir Y global
Info=list();
scs_m.props.tf=300; // temps final de simulation
YY=[];  // utilise pour le stockage des resultats
A=[.1:.1:5];  // valeurs possibles de a
for a=A
  %scicos_context.a=a;  // intialisation de a
  disp(a)
  Info=scicos_simulate(scs_m,Info,%scicos_context,"nw");
  YY=[YY,Y];  // stockage dans Y
end
scf(1);clf()
plot2d(A',YY')  // affichage de resultat
```

Fichier source : `scicos_diag/suspexe.sce`

Noter l'initialisation de a à travers la structure `%scicos_context` et l'utilisation du drapeau `''nw''` comme argument de la fonction `scicos_simulate`. Ce dernier

indique une simulation sans fenêtre ce qui implique dans ce cas la désactivation de `MScope` pour accélérer les simulations. L'argument `scs_m` contient le schéma complet qui est chargé quand un fichier `.cos` est chargé dans Scilab. Enfin, l'argument `Info` est un tableau de travail utilisé par la fonction `scicos_simulate` pour garder certaines informations (par exemple les résultats de la compilation) d'un appel sur l'autre.

Les courbes affichées par ce script sont les suivantes :



On y voit les valeurs de `crit 1` et `crit 2` en fonction de a. On peut facilement trouver la valeur optimale de a en examinant ces courbes. Dans des cas plus complexes et en particulier si on a plusieurs paramètres à ajuster, on utilisera les fonction `optim`, `leastsq`, ou `datafit` pour trouver les valeurs optimales des paramètres.

10.9 Conclusion

Nous avons présenté dans ce chapitre quelques exemples d'utilisation de Scicos et quelques recettes pour la construction des nouveaux blocs.

Nous avons aussi montré comment les simulations Scicos peuvent être lancées en mode batch par Scilab en utilisant la fonction `scicos_simulate`.

Il existe d'autres fonctions Scilab, plus spécialisées, manipulant des schémas Scicos comme la fonction `steadycos` qui calcule l'état stationnaire d'un schéma Scicos ou la fonction `lincos` qui le linéarise. Ces fonctions ne sont pas présentées ici mais elles sont documentées dans le "Help" de Scilab. Pour plus d'information, vous pouvez aussi consultez les documentations sur le site Web de Scicos, `www.scicos.org`, ou le livre [5].

Chapitre 11

Statistiques et Probabilités

11.1 Fonction de répartition empirique

Nous proposons ici quelques fonctionnalités de Scilab pour visualiser des données statistiques.

Pour visualiser la fonction de répartition empirique de données $((x_i)_{i=1,N})$:

$$F(u) = \frac{\text{card}\{i, x_i \leq u\}}{\text{card}\{x_i\}}$$

il suffit de tracer le graphe d'une fonction constante par morceaux passant par les couples de points $(y_i, i/N)_{i=1,N}$, le vecteur y étant obtenu par un tri croissant (fonction `gsort`) du vecteur de données x (voir la figure 11.1) :

On effectue ici un tirage d'un N-échantillon de loi gaussienne, Puis nous visualisons la fonction de répartition empirique :

```
//tirage
N=200;
x = rand(1,N,"n");

//viusalisation
plot2d2(gsort(x,"g","i")',(1:N)'/N)
e=gce();e.children.polyline_style=2;//staircase

xtitle("Fonction de répartition empirique")
```

Fichier source : `scilab/repemp.sce`

Pour chercher une valeur particulière de la fonction de répartition empirique, on utilise la fonction `find`. Par exemple, pour `u=0.56` on trouve la valeur correspondante de $F(u)$ par :

```
->u = 0.56
->F = size(find( x <= u),"*")/ N;
```

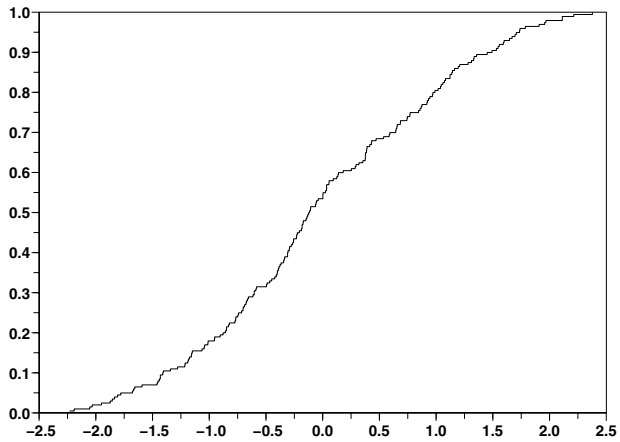


Figure 11.1 – Fonction de répartition empirique.

La fonction inverse de F, notée $Q = F^{-1}$ est aussi fort utile car elle permet notamment de calculer les quantiles de la distribution empirique des données. Voici un moyen de programmer la fonction inverse d'une fonction de répartition empirique. On définit une fonction `Q` qui utilise une variable globale `y` (qui n'est autre que les données triées en ordre croissant) :

```
->y=gsort(x,"g","i")';
->function z=Q(u),z=y(u*N),endfunction; // u compris dans (0,1)
```

Pour un certain nombre de lois de probabilité classiques, des fonctions Scilab permettent le calcul numérique des fonctions de répartition inverse. Ces fonctions sont généralement appelées `cdf*`, où `cdf` sont les initiales de *cumulative distribution function*. Par exemple, la fonction `cdfnor` nous permet d'obtenir un calcul numérique de la fonction de répartition inverse d'une loi normale. Nous encapsulons ici l'appel de `cdfnor` dans une fonction à la syntaxe plus courte `Qnor` :

```
function z=Qnor(u)
  z=cdfnor("X",0,1,u,1-u)
endfunction
```

Nous pouvons maintenant utiliser ce qui précède pour comparer les quantiles empiriques avec les quantiles d'une loi normale. On visualise sur un même graphe les couples de points constitués d'une part par les quantiles de la loi normale et d'autre part par les quantiles de la loi empirique des données.

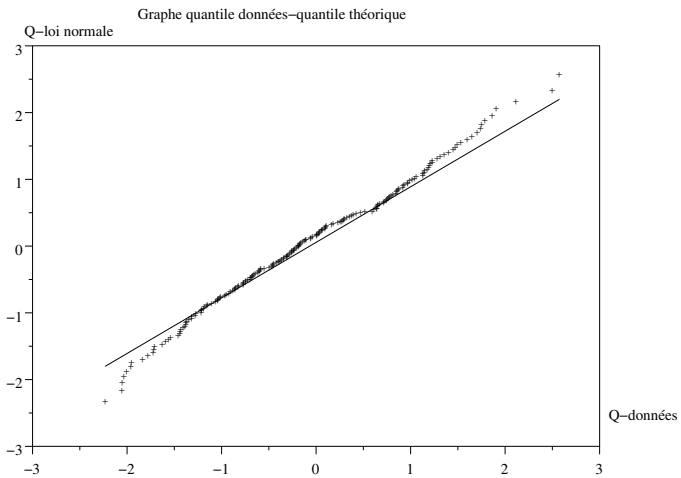


Figure 11.2 – Q-Q graphe.

On commence par évaluer les fonction `Q` et `Qnor` :

```
a=(1:N)/N;
b1=feval(a(2:$-1),Q);    // y=feval(x,f) donne y(i)=f(x(i))
b2=feval(a(2:$-1),Qnor);
```

Puis on dessine le graphe des couples `(b1(i),b2(i))` en rajoutant la droite qui passe par les quartiles. L'écart à la droite nous donne une idée graphique de la non-adéquation des données aux quantiles théoriques (voir la figure 11.2) :

```
plot2d(b1',b2',style=-1); // les couples (b1,b2)

function z=Qline(u)
 z= (u-Q(1/4))*(Qnor(3/4)-Qnor(1/4))/(Q(3/4)-Q(1/4))+Qnor(1/4)
endfunction

plot2d([min(b1);max(b1)],[Qline(min(b1));Qline(max(b1))]);

titre ="Graphe quantile données-quantile théorique";

xtitle(titre,"Q-données","Q-loi normale");
```

Dans ce qui précède, on peut remplacer les quantiles de la loi normale `Qnor(i/N)` par `Qnor((i-3/8)/(N+1/4))` ([17] page 101) pour tenir compte de corrections dues à la discrétisation.

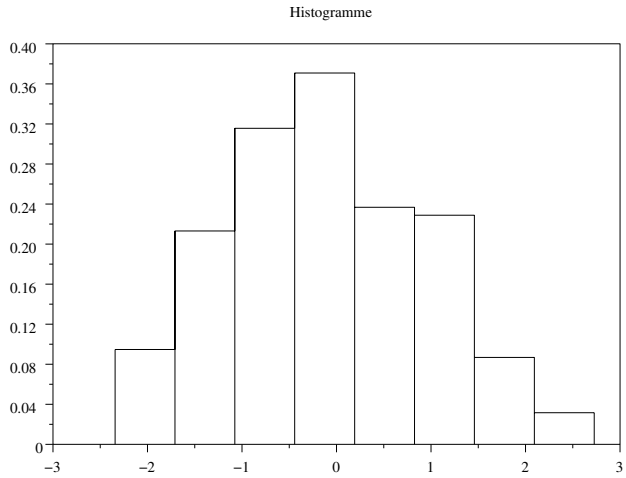


Figure 11.3 – Histogramme.

11.2 Histogrammes et densités de probabilité

Reprenons les données précédentes et cherchons cette fois à approcher la densité de probabilité de la variable aléatoire sous-jacente en regardant la loi de probabilité empirique des données. On peut commencer par un histogramme des données (voir la figure 11.3) obtenu au moyen de la fonction `histplot`. Le nombre de classes doit être choisi par l'utilisateur, nous utilisons ici la formule de Sturges :

```
->// nombre de classes (formule de Sturges)
->Nc=int(log(N)/log(2)+1);
->histplot(Nc,x);xtitle("Histogramme")
```

On peut chercher à obtenir une estimation de la densité par une fonction plus lisse. On peut faire cela au moyen de noyaux régularisants, par exemple en utilisant la formule suivante :

$$\hat{f}(u) = \frac{1}{Nh} \sum_{j=1}^{N} K(\frac{u - x_j}{h})$$

où $K(x)$ est une fonction que l'on choisit comme étant une densité de probabilité et h est un paramètre qui va contrôler le lissage. Le choix de h est délicat : s'il est trop petit on ne lisse pas assez, s'il est trop grand on risque de perdre des pics importants dans la distribution. [10, 16] Choisissons ici pour K une

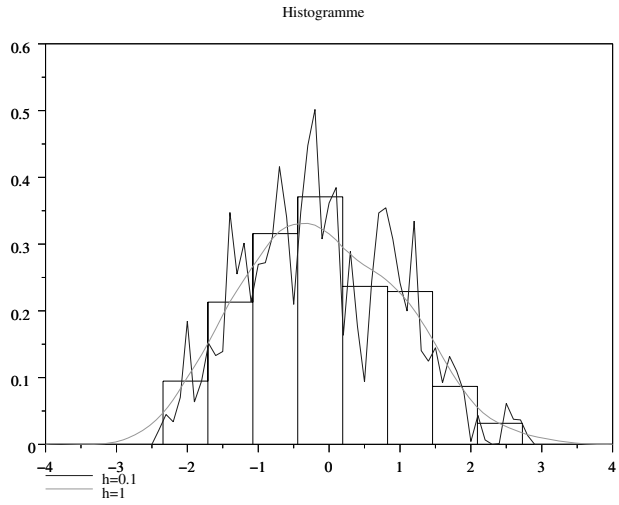


Figure 11.4 – Lissage avec un noyau triangle.

fonction triangle :

$$K(x) = \sup(1 - \text{abs}(x), 0)$$

Nous programmons ici une fonction `y=fchap(u,xdat,h)` qui, pour des entrées vectorielles, renvoie `y` tel que $y(i) = \hat{f}(u(i))$, `xdat` représentant nos données d'origine.

```
function y=fchap(u,xdat,h)
  u=u(:)              // on fait un vecteur colonne au cas ou
  n=size(xdat,"*")    // xdat doit être un vecteur ligne
  m=size(u,"*")
  y=1/h*maxi(0, 1- abs(u*ones(1,n) - ones(m,1)*xdat)/h);
  y=(1/n)*sum(y,"c") ;
endfunction
```

Fichier source : `scilab/fchap.sci`

On trace pour plusieurs valeurs de `h` la densité estimée (voir la figure 11.4) :

```
u=(-4:0.1:4)';
// On superpose les graphes a l'histogramme précédent.
plot2d(u,[fchap(u,x,0.1),fchap(u,x,1)],style=[2,3],..
       leg="h=0.1@h=1");
```

Fichier source : `scilab/fchap.sce`

11.3 Simulation de variables aléatoires

Plusieurs fonctions sont disponibles pour effectuer des tirages de n-échantillons d'une loi donnée. Pour effectuer un tirage de loi uniforme sur $[0,1]$ ou de loi normale on peut utiliser la fonction `rand`. Un autre générateur est disponible avec la fonction `grand` (qui interface une bibliothèque développée par Barry W. Brown et James Lovato du département de Biomathématiques de l'Université du Texas à Houston). La fonction `grand` permet d'effectuer des tirages pour de nombreuses lois. Par exemple effectuons `N` tirages de permutations (avec une loi uniforme) du vecteur `1:p`

```
->p=3;x=(1:p)';   // vecteur colonne des valeurs
->N=100;          // nombre de permutations souhaitées
->M=grand(N,"prm",x);
->// chaque colonne de M contient une permutation
->M(:,1)
 ans  =

!   2. !
!   3. !
!   1. !
```

Pour certaines lois, non présentes dans `grand`, on peut utiliser les fonctions `cdf*` vues précédemment qui donnent les fonctions de répartition empiriques et leurs inverses pour certaines lois. Par exemple, pour effectuer un tirage d'un n-échantillon de loi de Student à 9 degrés de liberté on pourra utiliser `cdft` et écrire :

```
N=10000; x=rand(1,N,"u");   // x:N tirage de loi uniforme (0,1)
// composition avec l'inverse de la fonction de répartition
// empirique de la loi de Student à 9 degrés de liberté
Ts=cdft("T",9*ones(x),x,1-x);
histplot(50,Ts); // histogramme des N tirages
```

11.4 Intervalles de confiance et tests

Les fonctions de répartition et leurs inverses implémentées dans Scilab permettent aussi d'effectuer des calculs d'intervalles de confiance et des tests.

Par exemple, soit X une variable aléatoire de loi normale dont on ignore la moyenne m et l'écart type σ. À partir d'un N-échantillon on peut estimer la moyenne par :

$$\overline{X} = \sum_{i=1}^{N} X_i/N$$

et la variance de X par :

$$s^2 = \sum_{i=1}^{N} (X_i - \overline{X})^2/(N-1)$$

On sait alors que $S = (\overline{X} - m)\sqrt{N}/s$ suit une loi de Student à $N-1$ degrés de liberté. On peut utiliser ce résultat pour trouver un intervalle de confiance symétrique et exact pour l'estimation de la moyenne par $\overline{X}$ pour un niveau de confiance α donné. La fonction `cdft` va nous permettre d'effectuer ces calculs.

```
->alpha=0.05; // seuil de confiance
->N=10;sigma=3;
->x= sigma *rand(1,N,"n"); // génération de données
->m = mean(x); // estimation de la moyenne
->s = st_deviation(x) // estimation de l'écart type.
// intervalle de confiance (quand on ne connaît pas sigma)
// au niveau 1-alpha
->C=cdft("T",N-1,1-alpha/2,alpha/2) // P(S <= C)= 1-alpha/2
 C  =

    2.2621572
->Ic=[m-C*s/sqrt(N),m, m+C*s/sqrt(N)]
 Ic  =

! - 1.1641292    0.6270794    2.4182879 !
// intervalle de confiance (sigma connu) au niveau alpha
->Cn=cdfnor("X",0,1,1-alpha/2,alpha/2)
 Cn  =

    1.959964

->Ic=[m- sqrt(1/N)*Cn*sigma,m, m+sqrt(1/N)*Cn*sigma]
 Ic  =

! - 1.2323057    0.6270794    2.4864644 !
```

L'intervalle de confiance doit se lire de la façon suivante : si l'on répète l'expérience précédente, la proportion d'intervalles de confiance contenant `m` doit être en moyenne de $(1-\alpha)$. Un autre test classique est le test du χ^2 que nous décrivons maintenant. Soient p_1,..., p_k, k nombres réels strictement positifs tels que $p_1 + \cdots + p_k = 1$, et soient X_1,..., X_n des variables aléatoires à valeurs dans $\{s_1, ..., s_k\}$ dont on souhaite tester si elles vérifient $\mathbb{P}(X_1 = s_j) = p_j$ $(j = 1, ..., k)$. On pose :

$$N_j^{(n)} = \sum_{i=1}^{n} \mathbb{I}_{\{X_i = s_j\}}, \quad j = 1, ..., k$$

et on appelle *statistique du* χ^2 la quantité :

$$T_n = \sum_{j=1}^{k} \frac{(N_j^{(n)} - np_j)^2}{np_j}.$$

On évalue avec la fonction `cdfchi` la probabilité qu'une variable aléatoire suivant une loi du χ^2 à $k-1$ degrés de liberté dépasse la valeur prise par la variable aléatoire T_n.

Effectuons tout d'abord un tirage de n valeurs d'une loi discrète donnée par deux vecteurs `p` et `s` de taille `k`. Pour ce faire, on effectue un tirage d'une trajectoire d'une chaîne de Markov dont la matrice de transition contient k fois la ligne `p`.

```
->n=100; // nombre de tirage
->p= [1:10]; p = p./sum(p);  // le vecteur p
->s= [0,1,4,8,9,13,16,20,21,45] ; //   vecteur s
->M = ones(p')*p ; // Matrice de chaîne de Markov
->X=grand(n,"markov",M,1);
->//passage des numéros d'état de la chaîne aux valeurs s
->X=s(X);
```

Calculons maintenant les $N_j^{(n)}$ au moyen de la fonction `find` puis la valeur de T_n :

```
->N=0*p;
->for i=1:size(s,"*"), N(i)= size(find(X==s(i)),"*"); end
->sum(N)  // on verifie que sum(N) == n
 ans  =

    100.
->Tn = sum((N - n*p).^2 ./(n*p))
 Tn  =

    8.1771429
```

On regarde la valeur correspondante pour une loi du χ^2 à $k-1$ degrés de liberté :

```
->k = size(p,"*")
->[O,P]= cdfchi("PQ",Tn,k-1)
 P  =              // probabilité de dépassement de Tn

    0.5163979
 O  =             //  1-P

    0.4836021
```

11.5 Analyse de la variance à un facteur pour harmoniser des notes d'examen

Des copies d'examen en nombre I sont réparties auprès de J correcteurs. Après correction, chaque correcteur rend ses copies et l'on constate que les moyennes des notes par correcteur sont différentes. Comment estimer si ce phénomène provient des correcteurs ou de la variabilité des copies ?

L'analyse de la variance à un facteur va nous permettre de construire un test pour répondre à cette question (voir par exemple [18] pour une démonstration des résultats exposés ici).

On dispose des notes fournies par chaque correcteur que l'on voit comme des réalisations d'une variable aléatoire Y. Le fait d'avoir J correcteurs se traduit par une variable qualitative que l'on appelle A et qui a J modalités notées $a_1, \ldots, a_J$. On suppose que, sur la sous-population des individus ayant la modalité a_j (c'est-à-dire la sous-population qui est corrigée par le correcteur j), la variable aléatoire Y prend la valeur (aléatoire) Y_j d'espérance mathématique $\mathbb{E}(Y_j) = m + \alpha_j$, où m (effet commun) et α_j (effet du correcteur j) sont des paramètres déterministes inconnus.

Plus précisément, sur tout individu i de cette sous-population d'indice j et de taille I_j :

$$Y_{i,j} = m + \alpha_j + u_{i,j} \quad i = 1, ..., I_j, \quad j = 1, ..., J,$$

où $I = \sum_{j=1}^{J} I_j$ est la taille totale de la population et les $u_{i,j}$ sont mutuellement indépendantes, de loi $N(0, \sigma)$, d'écart-type σ inconnu mais constant (indépendant de j et de i).

On se pose la question suivante : estimer les paramètres m et α_j, c'est-à-dire estimer la moyenne des copies pour chaque correcteur et tester l'hypothèse :

$$(H_A) \quad \forall j, \alpha_j = \text{constante}.$$

Cela veut dire que la moyenne des notes n'est pas influencée par les correcteurs.

On suppose que les notes de chaque correcteur sont données dans une liste `p`, de sorte que `p(j)` est un vecteur de taille I_j donnant les notes du correcteur j.

```
p= list([8.7,16,10.3,16.1,12.2,9.8,9.1,12.7,11.5,10.9,14,9.3,..
           13.7,9.1,8.8,12.5],..
        [8,10,12,16,11,14,13,10,12,8,16,8,11,8,12,11,14],..
        [8,14,17,14,15,14,12,10,14,16,17,12,12,10,8,12,11,12],..
        [12.5,11,11,12,11.5,11.5,14.5,13.5,12.5,14,14,13,10.5,..
           14,13]);
```

On parcourt la liste en effectuant les calculs du nombre d'individus de chaque classe I_j, de la moyenne de chaque classe et des écarts type de chaque population :

```
J=length(p) ; // nombre de modalités (ou de correcteurs)
Iv=[];  pmoy=[];sigma=[];
for pi=p,
   Iv=[Iv,size(pi,"*")]; pmoy=[pmoy, mean(pi)];
   sigma=[sigma,st_deviation(pi)];
end
I = sum(Iv); // nombre total d'individus.
->pmoy         // les moyennes de chaque correcteur.
 pmoy  =

!   11.54375    11.411765    12.666667    12.566667 !
```

On calcule `ssa` qui mesure la dispersion des moyennes obtenues pour chaque correcteur :

$$ssa = \sum_{j=1}^{J} I_j * (pmoy(j) - E)^2 \quad E = \frac{1}{J}\sum_{j=1}^{J} pmoy(j)$$

```
pmoymoy=mean(pmoy); // la moyenne des moyennes
ssa =  ((pmoy-pmoymoy).^2 )*Iv';
```

On calcule `ssr` qui mesure la dispersion à l'intérieur des sous-populations

$$ssr = \sum_{j=1}^{J} \left(\sum_{i=1}^{I_j} p(j)(i) - pmoy(j) \right)^2$$

```
 ssr=0;
 for i=1:J; ssr= ssr + norm(p(i) -pmoy(i))**2; end;
```

Sous l'hypothèse (H_A), `f=(ssa/J-1)/(ssr/I-J)` est un tirage selon une loi de Fisher-Snedecor de paramètres $(J-1, I-J)$. Si l'hypothèse (H_A) n'est pas vérifiée les calculs montrent que f est alors supérieur à 1. On peut donc baser le rejet de l'hypothèse H_A sur un test unilatéral. Pour un seuil de confiance donné α, on calcule F_A tel que $\mathbb{P}(\{F \geq F_A\}) \leq \alpha$, F suivant une loi de Fisher-Snedecor $(J-1, I-J)$:

```
alpha=0.05;
f=(ssa/(J-1))/(ssr/(I-J))
 f  =

    1.2967175
Fa=cdff("F",J-1,I-J,1-alpha,alpha)   // Calcul de Fa
 Fa  =

    4.114194
[P,Q]=cdff("PQ",f,J-1,I-J);  // seuil associe a f
```

```
->Q
 Q  =

    0.2835308
```

On trouve `f < Fa` et donc au seuil de confiance de 5% on ne rejette pas l'hypothèse H_A d'égalité des moyennes. Le deuxième calcul $Q = 0.283$, nous indique qu'il faudrait fixer un seuil de confiance de valeur Q pour rejeter l'hypothèse H_A.

11.6 Régression linéaire

On suppose ici que le modèle auquel on s'intéresse s'écrit sous la forme :

$$Y_i = \sum_{k=0}^{p-1} b_k x_{k,i} + E_i$$

où les variables aléatoires $(E_i)_{i=1,n}$ vérifient les conditions de Gauss-Markov (c'est-à-dire $\mathbb{E}(E_i) = 0$, $\mathbb{E}(E_i^2) = \sigma^2$, $\mathbb{E}(E_i E_j) = 0$ pour $i \neq j$). On veut estimer les paramètres b_i à partir de n valeurs mesurées du $(p+1)$-uplet $(y_1, x_0, \ldots, x_{p-1})$. On notera que, pour obtenir un terme constant dans la régression linéaire, il suffit de choisir pour x_0 un vecteur dont tous les éléments valent 1.

On retient comme estimateur de b la valeur $\hat{b}$ qui minimise l'écart quadratique entre les Y mesurés et les Y prédits, et qui s'écrit :

$$\hat{b} = \underset{b}{\text{Argmin}} \sum_{i=1}^{n} \left(y_i - \sum_{k=0}^{p} b_k x_{ki} \right)^2 .$$

En notant $Y = (y_1, \ldots, y_n)^T$, $x_k = (x_{k1}, \ldots, x_{kn})^T$, et $X = (x_1, \ldots, x_n)$, la solution du problème s'écrit de façon vectorielle $\hat{b} = (X^T X)^{-1} X^T Y$.

```
->n=20;
->p=3;
->X=[ones(n,1),rand(n,2)];
->sigma=0.1;
->Y=X*[1;2;3] + sigma*rand(n,1,"n");
```

On obtient les coefficients de la régression linéaire en appliquant la formule précédente

```
->M = inv(X'*X);
->B = M * X'*Y
!   0.8831196 !
!   2.1253564 !
!   3.1110742 !
```

ou plus simplement, en utilisant l'opérateur « \ » qui permet de résoudre un système linéaire, la résolution se faisant au sens des moindres carrés pour $n > p$:

```
->B1 = X\Y
 B1  =

!   0.8831196 !
!   2.1253564 !
!   3.1110742 !
```

On peut ensuite estimer la quantité σ^2 en utilisant le résidu `Res`, différence entre les Y mesurés et ceux calculés par le modèle (`X*B`). L'estimateur de σ^2 est noté `s2` :

```
->Ycalcule =  X * B;// Y calculés par le modèle
->Res = Y - Ycalcule; // Résidus non expliqués
->s2 = Res'*Res/(n-p)  // estimation de sigma^2
 s2  =

    0.0100629
```

On peut aussi estimer la covariance de l'estimateur $\hat{b}$ notée `covB` et l'écart type sur chaque estimateur `sigB` :

```
->covB = s2*M
 covB  =

!   0.0042571  - 0.0032525  - 0.0048620 !
! - 0.0032525    0.0065939  - 0.0000876 !
! - 0.0048620  - 0.0000876    0.0111941 !

->sigB = sqrt(diag(covB)) // estimation des écarts types des B
 sigB  =

!   0.0652468 !
!   0.0812027 !
!   0.1058023 !
```

Sous des hypothèses de normalité des résidus, les paramètres à estimer sont gaussiens et $T_j = B_j/sigB$ suit une loi de Student à $(n-p)$ degrés de liberté. On peut, pour chaque paramètre estimé, obtenir la valeur T_j et la probabilité associée (`Qr(j)`) pour que le module de T, où T suit une loi de Student à $(n-p)$ degrés de liberté, dépasse la valeur $|T_j|$:

```
->T = B./sigB

->// Qr1(j) = P(T > |T_j|)
->[Pr1,Qr1] = cdft("PQ",abs(T),(n-p)*ones(T));
```

```
->// Pr2(j) = P(T < -|T(j)|)
->[Pr2,Qr2] = cdft("PQ",-abs(T),(n-p)*ones(T));
->Pr = Pr2+Qr1
 Pr  =

   1.0E-12 *

!   156.28169 !
!   0.0035463 !
!   0.0005105 !
```

Le calcul précédent nous permet d'obtenir un intervalle de confiance sur les valeurs estimées des paramètres en se fixant un seuil de confiance :

```
->alpha=0.05;
->C=cdft("T",n-2,1-alpha/2,alpha/2) // P( T <= C)= 1-alpha/2
->Ic = [B - sigB*C, B + sigB*C]
 Ic  =

!   0.7460412    1.0201979 !
!   1.954756     2.2959569 !
!   2.8887918    3.3333565 !
```

Dans l'exemple précédent on notera que le seuil de confiance $\alpha = 5\%$ n'est valable que si l'on cherche un seuil de confiance sur un seul des paramètres. Le seuil de confiance sur les trois paramètres simultanés est de $1-(0.95)^3$ soit environ 14%. On peut en revanche utiliser le fait que :

$$(B - \mathbb{E}(B))' \texttt{covB}^{-1} (B - \mathbb{E}(B))$$

suit une loi du χ^2 à 3 degrés de liberté pour trouver un ellipsoïde de confiance donnée par :

$$(B - \mathbb{E}(B))^T \texttt{covB}^{-1} (B - \mathbb{E}(B)) \leq k$$

au seuil de confiance $\alpha = 5\%$. La valeur de k s'obtient comme suit :

```
->k=cdfchi("X",3,0.95,0.05)
 ans  =

    7.8147279
```

Les calculs statistiques précédents utilisent la normalité des ε_i. On pourra tester cette normalité à travers les résidus en utilisant une représentation graphique comme dans la figure 11.2 ou en utilisant des tests. Par exemple dans les problèmes issus de série chronologiques le coefficient de Durbin et Watson `DW` permet de tester une hypothèse de corrélation des bruits ε_i de la forme $\varepsilon_{i+1} = \rho\varepsilon_i + n_i$ où les n_i sont cette fois non corrélés. Le test de Durbin-Watson permet de tester $\rho = 0$ contre son alternative $\rho \neq 0$. Une valeur proche de 2 de `DW` indique une valeur de ρ proche de zéro et donc une non-corrélation vraisemblable des bruits :

```
dRes = Res(2:$)-Res(1:$-1)
dw_s3 = dRes'*dRes ;
dw_s2 = Res'*Res ;
dw = dw_s3/dw_s2
 dw  =

    1.7236965
```

On peut obtenir une estimation de la loi de DW sous l'hypothèse $\rho = 0$ par simulation. Cela permet de construire un test en estimant par exemple `P`$=\mathbb{P}\{dw \leq DW \leq 4 - dw\}$. On construit donc des résidus par simulation à partir de tirage numérique sur les ε_i qui sont gaussiens sous l'hypothèse $\rho = 0$:

```
->M1= - X*M*X';M1= eye(M1)+M1;
->m=1000;
->eps = s2*rand(n,m,"n");
->e = M1*eps;
->dres = e(2:$,:)-e(1:$-1,:);
->dw_s3= diag(dres'*dres);
->dw_s2= diag(e'*e);
->DW = dw_s3./ dw_s2;
```

On dispose maintenant de m tirages de DW dans le vecteur `DW` et on calcule `P` sur la distribution empirique des m tirages :

```
->P1 = size(find( DW <= dw ),"*")/ m ;
->P2 = size(find( DW <= 4- dw ),"*")/ m ;
->P = P2 -P1
 P  =

    0.479
```

La régression linéaire sert pour la prévision ou pour les tests d'hypothèses sur les paramètres du modèle. Pour une valeur donnée des paramètres $x = (x_0, \ldots, x_n)^T$, le modèle de régression linéaire nous permet d'estimer la valeur de Y associée et de donner un intervalle de confiance sur cette valeur. Nous procédons comme suit :

```
->x0=[1;3.4;5.7];
->y0=x0'*B
 y0  =

    25.842454

->sig= sqrt(x0'*covB*x0)
->alpha=0.05;
// Utilisation de la loi de Student a n-p degrés de liberté
->C=cdft("T",n-p,1-alpha/2,alpha/2) // P( T <= C)= 1-alpha/2
->Ic = [y0 - sig*C, y0 + sig*C] // intervalle de confiance
```

```
 Ic  =

!   24.570878    27.11403 !
```

Annexes

Bibliographie

[1] G. Allaire and S. M. Kaber. *Algèbre linéaire numérique. Cours et exercices.* Ellipses, Paris, 2002.

[2] G. Allaire and S. M. Kaber. *Introduction à Scilab - Exercices pratiques corrigés d'algèbre linéaire.* Ellipses, Paris, 2002.

[3] P. N. Brown, A. C. Hindmarsh, and L. R. Petzold. Using Krylov methods in the solution of large-scale differential-algebraic systems. *SIAM J. Sci. Comp.*, 15 :1467–1488, 1994.

[4] C. Bunks, J.-P. Chancelier, F. Delebecque, C. Gomez, M. Goursat, R. Nikoukhah, and S. Steer. *Engineering and Scientific Computing with Scilab.* Birkhäuser, 1999.

[5] S. L. Campbell, J.-P. Chancelier, and R. Nikoukhah. *Modeling and Simulation in Scilab/Scicos.* Springer Verlag, 2005.

[6] J.-P. Chancelier, F. Delebecque, C. Gomez, M. Goursat, R. Nikoukhah, and S. Steer. *Introduction à Scilab (première édition).* Springer, 2001.

[7] S. Guerre-Delabriere and M. Postel. *Méthodes d'approximation - Équations différentielles - Applications Scilab. Niveau L3.* Ellipses, Paris, 2004.

[8] A. C. Hindmarsh. lsode and lsodi, two new initial value ordinary differential equation solvers. *ACM-Signum Newsletter*, 15(4) :10–11, 1980.

[9] A. C. Hindmarsh. *ODEPACK, A Systematized Collection of ODE Solvers, in Scientific Computing.* vol. 1 of IMACS Transactions on Scientific Computation. North-Holland, Amsterdam, 1983.

[10] W. Härdle. *Smoothing techniques with implementation in S.* Springer Verlag, 1991.

[11] J. Hubbard and B. West. *Équations différentielles et systèmes dynamiques (traduction et adaptation : V. Gautheron).* Cassini, 1999.

[12] D. Libes. *Exploring Expect.* O'Reilly and Associates, 1995.

[13] J. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[14] P. S. Motta Pires and D. A. Rogers. Free/open source software : An alternative for engineering students. In *Proc. 32nd ASEE/IEEE Frontiers in Education Conference*, pages T3G–7 TO 11, Boston, MA, 2002.

[15] K. Radhakrishnan and A. C. Hindmarsh. Description and use of LSODE, the livermore solver for ordinary differential equations. Technical Report report UCRL-ID-113855, Lawrence Livermore National Laboratory, December 1993.

[16] D. W. Scott. *Multivariate density estimation. Theory, Practice and Visualization*. John Wiley and sons, 1992.

[17] A. Sen and M. Srivastava. *Regression Analysis. Theory, Methods and Applications*. Springer Verlag, 1990.

[18] K. S. Trivedi. *Probability and Statistics with reliability queuing and computer science application*. Prentice-Hall Englewood Cliffs, N.J., 1982.

[19] G. E. Urroz. Numerical and statistical methods with scilab for science and engineering.

[20] B. Welch. *Practical Programming in Tcl and Tk*. Prentice-Hall, 1997.

[21] A. Yger. *Théorie et analyse du signal. Cours et initiation pratique via Matlab et Scilab*. Ellipses, Paris, 1999.

Index

Achevé d'imprimer sur les presses de l'Imprimerie BARNÉOUD
B.P. 44 - 53960 BONCHAMP-LÈS-LAVAL
Dépôt légal : juin 2007 - N° d'imprimeur : 701082
Imprimé en France