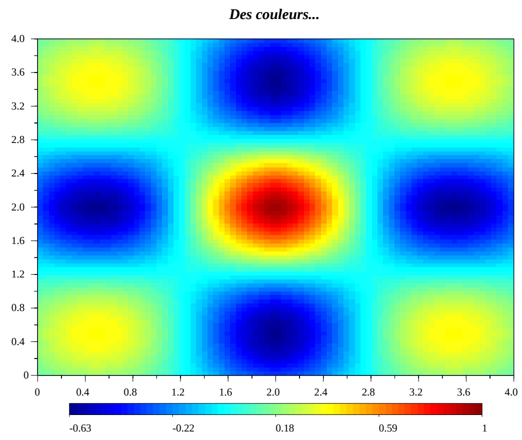
Une introduction à Scilab

version 0.996



Bruno Pinçon

Institut Elie Cartan Nancy
E.S.I.A.L.
Université Henri Poincaré
Email : `Bruno.Pincon@iecn.u-nancy.fr`

Ce document a été initialement rédigé pour les étudiants ingénieurs de l'E.S.I.A.L. (École Supérieure d'Informatique et Application de Lorraine). Il ne décrit qu'une petite partie des possibilités de Scilab, essentiellement celles qui permettent la mise en pratique des rudiments d'analyse numérique que je leur enseigne, c'est à dire :

– la manipulation des matrices et vecteurs de nombres flottants ;
– la programmation en Scilab ;
– quelques primitives graphiques.

Scilab permet de faire beaucoup d'autres choses, en particulier dans le domaine de l'automatique, du traitement du signal, de la simulation de systèmes dynamiques (avec scicos)... Comme je pense compléter progressivement ce document, je suis ouvert à toutes remarques, suggestions et critiques permettant de l'améliorer (même sur les fautes d'orthographe...), envoyez les moi par Email.

Remerciements : au Doc Scilab qui m'a souvent aidé via le forum des utilisateurs, à Bertrand Guiheneuf qui m'a fourni le « patch » magique pour compiler Scilab 2.3.1 sur ma linuxette (la compilation des versions suivantes ne pose pas de problème sous linux), et à mes collègues et amis, Stéphane Mottelet[1], Antoine Grall, Christine Bernier-Katzentsev et Didier Schmitt. Finalement, un grand merci à Patrice Moreaux pour sa relecture attentive et les corrections dont il m'a fait part.

1. merci pour les « trucs » pdf Stéphane !

Table des matières

Chapitre 1

Informations diverses

1.1 Scilab en quelques mots

Qu'est-ce que Scilab ? Soit vous connaissez déjà Matlab et alors une réponse rapide consiste à dire que Scilab en est un pseudo-clone libre (voir un plus loin quelques précisions à ce sujet) développé[1] par l'I.N.R.I.A. (Institut National de Recherche en Informatique et Automatique). Il y a quand même quelques différences mais la syntaxe est à peu près la même (sauf en ce qui concerne les graphiques). Si vous ne connaissez pas Matlab alors je vais dire brièvement que Scilab est un environnement agréable pour faire du calcul numérique car on dispose sous la main des méthodes usuelles de cette discipline, par exemple :

- résolution de systèmes linéaires (même creux),
- calcul de valeurs propres, vecteurs propres,
- décomposition en valeurs singulières, pseudo-inverse
- transformée de Fourier rapide,
- plusieurs méthodes de résolution d'équations différentielles (raides / non raides),
- plusieurs algorithmes d'optimisation,
- résolution d'équations non-linéaires,
- génération de nombres aléatoires,
- de nombreuses primitives d'algèbre linéaire utiles pour l'automatique.

D'autre part, Scilab dispose aussi de toute une batterie d'instructions graphiques, de bas-niveau (comme tracer un polygone, récuperer les coordonnées du pointeur de la souris, etc...) et de plus haut niveau (pour visualiser des courbes, des surfaces) ainsi que d'un langage de programmation assez simple mais puissant et agréable car il intègre les notations matricielles. Quand vous testez un de vos programmes écrit en langage Scilab, la phase de mise au point est généralement assez rapide car vous pouvez examiner facilement vos variables : c'est comme si l'on avait un débogueur. Enfin, si les calculs sont trop longs (le langage est interprété...) vous pouvez écrire les passages fatidiques comme des sous-programmes C ou fortran (77) et les lier à Scilab assez facilement.

1.2 Comment utiliser ce document

Rendez-vous en premier au chapitre deux où j'explique comment utiliser Scilab comme une calculette matricielle : il suffit de suivre les exemples proposés. Vous pouvez passer les sections étoilées (*) dans une première lecture. Si vous êtes intéressé(e) par les aspects graphiques vous pouvez alors essayer les premiers exemples du chapitre quatre. Le chapitre trois explique les rudiments de la programmation en Scilab. J'ai commencé à écrire un chapitre cinq concernant quelques applications ainsi qu'un « bétisier » qui essaie de répertorier les erreurs habituelles que l'on peut commettre en Scilab (envoyer moi les votres !). Une dernière chose, l'environnement graphique de Scilab (la fenêtre principale, les fenêtres graphiques,...) est légèrement différent entre ses deux versions Unix et Windows (cette dernière étant un portage de

1. en fait Scilab utilise de nombreuses routines qui proviennent un peu de partout et qui sont souvent accessibles via Netlib

la version « principale » développée sous Unix), c-a-d que les boutons et menus ne sont pas agencés de la même manière. Dans ce document certains détails (du genre sélectionner l'item « truc » du menu « bidule »...) sont relatifs à la version Unix mais vous trouverez sans problème la manipulation équivalente sous Windows.

1.3 Principe de travail sous Scilab

Au tout début Scilab peut s'utiliser simplement comme une calculette capable d'effectuer des opérations sur des vecteurs et matrices de réels et/ou complexes (mais aussi sur de simples scalaires) et de visualiser graphiquement des courbes et surfaces. Dans ce cas basique d'utilisation, vous avez uniquement besoin du logiciel Scilab. Cependant, assez rapidement, on est amené à écrire des scripts (suite d'instructions Scilab), puis des fonctions et il est nécessaire de travailler de pair avec un éditeur de texte comme par exemple, emacs (sous Unix et Windows), wordpad (sous Windows), ou encore nedit, vi (sous Unix)...

1.4 Où trouver de l'information sur Scilab?

La suite du document suppose que vous avez à votre disposition la version 2.3, 2.3.1, 2.4, 2.4.1 ou mieux 2.5 du logiciel. Pour tout renseignement consulter la « Scilab home page » :

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

à partir de laquelle vous avez en particulier accès à différentes documentations, aux contributions des utilisateurs, etc...

Le « Scilab Group » écrit depuis le mois de Décembre 99 un article mensuel dans « Linux magazine ». Plusieurs aspects de Scilab (dont la plupart ne sont pas évoqués dans cette introduction) y sont présentés, je vous les recommande donc.

Scilab dispose aussi d'un forum usenet qui est le lieu adéquat pour poser des questions, faire des remarques, rapporter un bug, apporter une solution à une question préalablement posée, etc... :

`comp.sys.math.scilab`

Tous les messages qui ont été postés dans ce forum sont archivés et accessibles à partir de la « home page » Scilab.

Pour terminer, je signale un nouveau document intéressant « Scilab Bag Of Tricks » élaboré par Lydia E. van Dijk et Christoph L. Spiel que vous trouverez à l'adresse :

`http://www.lightlink.com/lydia/sci-bot/book1.htm/`

et qui est disponible sous plusieurs formats (HTML, SGML, postscript, PDF).

1.5 Quel est le statut du logiciel Scilab?

Ceux qui connaissent bien les logiciels libres (généralement sous licence GPL) peuvent s'interroger sur le statut de Scilab en tant que logiciel « libre et gratuit ». Voici ce qu'en dit le Doc dans un message posté sur le forum :

> *Scilab: is it really free?*
>
> Yes it is. Scilab is not distributed under GPL or other standard free software copyrights (because of historical reasons), but Scilab is an Open Source Software and is free for academic and industrial use, without any restrictions. There are of course the usual restrictions concerning its redistribution; the only specific requirement is that we ask Scilab users to send us a notice (email is enough). For more details see Notice.ps or Notice.tex in the Scilab package.
>
> Answers to two frequently asked questions: Yes, Scilab can be included a commercial package (provided proper copyright notice is included). Yes, Scilab can be placed on commercial CD's (such as various Linux distributions).

Chapitre 2

La manipulation des matrices et vecteurs

Cette première partie donne des éléments pour commencer à utiliser Scilab comme une calculette matricielle

Pour lancer Scilab, il suffit de rentrer la commande[1] :

`scilab`

Si tout se passe bien, la fenêtre Scilab apparaît à l'écran avec en haut un menu (donnant en particulier accès au `Help` et aux `Demos`) suivi de la bannière scilab et de l'invite (`-->`) qui attend vos commandes :

```
                           ===========
                           S c i l a b
                           ===========


                          scilab-2.5
                   Copyright (C) 1989-99 INRIA


Startup execution:
  loading initial environment

-->
```

2.1 Entrer une matrice

Un des types de base de Scilab est constitué par les matrices de nombres réels ou complexes (en fait des nombres « flottants »). La façon la plus simple de définir une matrice (ou un vecteur, ou un scalaire qui ne sont que des matrices particulières) dans l'environnement Scilab est d'entrer au clavier la liste de ses éléments, en adoptant les conventions suivantes :

- les éléments d'une même ligne sont séparés par des espaces ou des virgules ;
- la liste des éléments doit être entourée de crochets [ ];
- chaque ligne, sauf la dernière, doit se terminer par un point-virgule.

Par exemple, la commande :

```
-->A=[1 1 1;2 4 8;3 9 27]
```

produit la sortie :

```
 A  =
```

1. sous Unix il faut bien sûr que la variable `PATH` contienne le chemin d'accès au logiciel ; sur Castor, Pollux et Océanos, ce chemin doit être `/usr/local/logiciel/scilab/bin` aller vérifier puis rajouter le dans votre fichier `.login`

```
!   1.    1.    1.  !
!   2.    4.    8.  !
!   3.    9.    27. !
```

mais la matrice est bien sûr gardée en mémoire pour un usage ultérieur. En fait si vous terminez l'instruction par un point virgule, le résultat n'apparaît pas à l'écran. Essayer par exemple :

```
-->b=[2 10 44 190];
```

pour voir le contenu du vecteur ligne b, on tape simplement :

```
-->b
```

et la réponse de Scilab est la suivante :

```
 b =

!  2.  10.  44.  190. !
```

Une instruction très longue peut être écrite sur plusieurs lignes en écrivant trois points à la fin de chaque ligne à poursuivre :

```
-->T = [ 1 0 0 0 0 0 ;...
-->      1 2 0 0 0 0 ;...
-->      1 2 3 0 0 0 ;...
-->      1 2 3 0 0 0 ;...
-->      1 2 3 4 0 0 ;...
-->      1 2 3 4 5 0 ;...
-->      1 2 3 4 5 6 ]
```

ce qui donne :

```
 T  =

!   1.    0.    0.    0.    0.    0. !
!   1.    2.    0.    0.    0.    0. !
!   1.    2.    3.    0.    0.    0. !
!   1.    2.    3.    4.    0.    0. !
!   1.    2.    3.    4.    5.    0. !
!   1.    2.    3.    4.    5.    6. !
```

Pour rentrer un nombre complexe, on utilise la syntaxe suivante (on peut se passer des crochets [] pour rentrer un scalaire) :

```
-->c=1 + 2*%i
 c  =

   1. + 2.i

-->Y = [ 1 + %i , -2 + 3*%i ; -1 , %i]
 Y  =

!   1. + i    - 2. + 3.i !
! - 1.          i        !
```

2.2 Quelques matrices et vecteurs types

Il existe des fonctions pour construire des matrices et vecteurs types, dont voici une première liste (il y en a bien d'autres dont nous parlerons ultérieurement ou que vous découvrirez avec le Help) :

- Pour obtenir une matrice identité de dimension (4,4) :

```
-->I=eye(4,4)
 I  =
```



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

Les arguments de la fonction `eye(n,m)` sont le nombre de lignes n et le nombre de colonnes m de la matrice (*Rmq :* si $n < m$ (resp. $n > m$) on obtient la matrice de la surjection (resp. injection) canonique de $\mathbb{K}^m$ vers $\mathbb{K}^n$.)

- Pour obtenir une matrice diagonale, dont les éléments diagonaux sont les composantes d'un vecteur :

```
-->B=diag(b)
 B  =

!   2.    0.     0.     0.   !
!   0.    10.    0.     0.   !
!   0.    0.     44.    0.   !
!   0.    0.     0.     190. !
```

(*Rmq :* cet exemple illustre le fait que Scilab distingue minuscule et majuscule, taper `b` pour vous rendre compte que ce vecteur existe toujours dans l'environnement). Appliquée sur une matrice la fonction `diag` permet d'en extraire sa diagonale principale sous la forme d'un vecteur colonne :

```
-->b=diag(B)
 b  =

!   2.   !
!   10.  !
!   44.  !
!   190. !
```

Cette fonction admet aussi un deuxième argument optionnel (cf exercices).

- Les fonctions `zeros` et `ones` permettent respectivement de créer des matrices nulles et des matrices « de 1 ». Comme pour la fonction `eye` leurs arguments sont le nombre de lignes puis de colonnes désirées. Exemple :

```
-->C = ones(3,4)
 C  =

!   1.    1.    1.    1. !
!   1.    1.    1.    1. !
!   1.    1.    1.    1. !
```

Mais on peut aussi utiliser comme argument le nom d'une matrice déjà définie dans l'environnement et tout se passe comme si l'on avait donné les deux dimensions de cette matrice :

```
-->O = zeros(C)
 O  =

!   0.    0.    0.    0. !
!   0.    0.    0.    0. !
!   0.    0.    0.    0. !
```

- Les fonctions `triu` et `tril` permettent elles d'extraire respectivement la partie triangulaire supérieure (u comme upper) et inférieure (l comme lower) d'une matrice, exemple :

```
-->U = triu(C)
 U  =
```

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

– La fonction `rand` (dont nous reparlerons)
permet de créer des matrices remplies de nombres pseudo-aléatoires (suivants une loi uniforme sur [0,1[mais il est possible d'obtenir une loi normale et aussi de choisir le germe de la suite) :

```
-->M = rand(2, 6)
 M  =

!   0.2113249    0.0002211    0.6653811    0.8497452    0.8782165    0.5608486 !
!   0.7560439    0.3303271    0.6283918    0.6857310    0.0683740    0.6623569 !
```

– Pour rentrer un vecteur (ligne) x à n composantes régulièrement réparties entre x_1 et x_n (c-à-d telles $x_{i+1} - x_i = \frac{x_n - x_1}{n-1}$, n « piquets » donc $n-1$ intervalles…), on utilise la fonction `linspace` :

```
-->x = linspace(0,1,11)
 x  =

!   0.    0.1    0.2    0.3    0.4    0.5    0.6    0.7    0.8    0.9    1. !
```

– Une instruction analogue permet, partant d'une valeur initiale pour la première composante, d'imposer « l'incrément » entre deux composantes, et de former ainsi les autres composantes du vecteur jusqu'à ne pas dépasser une certaine limite :

```
-->y = 0:0.3:1
 y  =

!   0.    0.3    0.6    0.9 !
```

La syntaxe est donc : `y = valeur_initiale:incrément:limite_a_ne_pas_dépasser`. Lorsque l'on travaille avec des entiers, il n'y pas de problème (sauf entiers très grands...) à fixer la limite de sorte qu'elle corresponde à la dernière composante :

```
-->i = 0:2:12
 i  =

!   0.    2.    4.    6.    8.    10.    12. !
```

Pour les réels (approché par des nombres flottants) c'est beaucoup moins évident du fait :

(i) que l'incrément peut ne pas « tomber juste » en binaire (par exemple $(0.2)_{10} = (0.00110011\ldots)_2$) et il y a donc un arrondi dans la représentation machine,

(ii) et des erreurs d'arrondi numérique qui s'accumulent au fur et à mesure du calcul des composantes.

Par exemple :

```
-->xx = 0:0.05:0.60
 ans  =

! 0.    0.05    0.1    0.15    0.2    0.25    0.3    0.35    0.4    0.45    0.5    0.55 !
```

Rmq : selon l'unité d'arithmétique flottante de l'ordinateur vous pouvez obtenir un résultat différent (c-à-d avec 0.6 comme composante supplémentaire). Souvent l'incrément est égal à 1 et on peut alors l'omettre :

```
-->ind = 1:5
 ind  =

!   1.    2.    3.    4.    5. !
```

Finalement, si l'incrément est positif (resp. négatif) et que `limite < valeur_initiale` (resp. `limite > valeur_initiale`) alors on obtient un vecteur sans composantes (!) qui est un objet Scilab appelée matrice vide (cf section quelques primitives matricielles supplémentaires) :

```
-->i=3:-1:4
```



```
 i  =

     []

-->i=1:0
 i  =

     []
```

2.3 L'instruction d'affectation de Scilab et les expressions scalaires et matricielles

Scilab est un langage qui possède une syntaxe simple (cf chapitre suivant) dont l'instruction d'affectation prend la forme :

`variable = expression`

ou plus simplement

`expression`

où dans ce dernier cas la valeur de `expression` est affectée à une variable par défaut `ans`. Une expression Scilab peut être toute simple en ne mettant en jeu que des quantités scalaires comme celles que l'on trouve dans les langages de programmation courants, mais elle peut aussi être composée avec des matrices et des vecteurs ce qui déroute souvent le débutant dans l'apprentissage de ce type de langage. Les expressions « scalaires » suivent les règles habituelles : pour des opérandes numériques (réels, complexes) on dispose des 5 opérateurs `+`, `-`, `*` , `/` et `^` (élévation à la puissance) et d'un jeu de fonctions classiques (cf table (2.1) pour une liste non exaustive). Noter que les fonctions liées à la fonction Γ ne sont disponibles que depuis la version 2.4 et que Scilab propose aussi d'autres fonctions spéciales parmi lesquelles on peut trouver des fonctions de Bessel, des fonctions elliptiques, etc…

Ainsi pour rentrer une matrice « coefficients par coefficients » on peut utiliser en plus des constantes (qui sont en fait des expressions basiques) n'importe quelle expression délivrant un scalaire (réel ou complexe), par exemple :

```
-->M = [sin(%pi/3) sqrt(2) 5^(3/2) ; exp(-1) cosh(3.7) (1-sqrt(-3))/2]
 M  =

!   0.8660254    1.4142136    11.18034          !
!   0.3678794    20.236014    0.5 - 0.8660254i !
```

(*Rmq :* cet exemple montre un danger potentiel : on a calculé la racine carrée d'un nombre négatif mais Scilab considère alors que l'on a affaire à un nombre complexe et renvoie l'une des deux racines comme résultat).

2.3.1 Quelques exemples basiques d'expressions matricielles

Toutes les opérations usuelles sur les matrices sont disponibles : somme de deux matrices de mêmes dimensions, produit de deux matrices (si leurs dimensions sont compatibles $(n,m) \times (m,p)$ …), produit d'un scalaire et d'une matrice, etc… Voici quelques exemples (pour lesquels on utilise une partie des matrices précédemment rentrées). *Rmq :* tout texte rentré sur une ligne après `//` est un commentaire pour Scilab : ne les tapez pas, ils sont là pour fournir quelques remarques et explications !

```
-->D = A + ones(A)   // taper A au préalable pour revoir le contenu de cette matrice
 D  =

!   2.    2.     2.  !
!   3.    5.     9.  !
!   4.    10.    28. !

-->A + M             // somme de matrice impossible (3,3) + (2,6) : que dit Scilab ?
```

abs	valeur absolue ou module
exp	exponentielle
log	logarithme népérien
log10	logarithme base 10
cos	cosinus (argument en radian)
sin	sinus (argument en radian)
tan	tangente (argument en radian)
cotg	cotangente (argument en radian)
acos	arccos
asin	arcsin
atan	arctg
cosh	ch
sinh	sh
tanh	th
acosh	argch
asinh	argsh
atanh	argth
sqrt	racine carrée
floor	partie entière $E(x) = (\lfloor x \rfloor) = n \Leftrightarrow n \leq x < n+1$
ceil	partie entière supérieure $\lceil x \rceil = n \Leftrightarrow n-1 < x \leq n$
int	partie entière anglaise : $int(x) = \lfloor x \rfloor$ si $x > 0$ et $= \lceil x \rceil$ sinon
erf	fonction erreur $erf(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$
erfc	fonction erreur complémentaire $ercf(x) = 1 - erf(x) = \frac{2}{\sqrt{\pi}} \int_x^{+\infty} e^{-t^2} dt$
gamma	$\Gamma(x) = \int_0^{+\infty} t^{x-1} e^{-t} dt$
lngamma	$\ln(\Gamma(x))$
dlgamma	$\frac{d}{dx} \ln(\Gamma(x))$

TAB. 2.1 – *quelques fonctions usuelles de Scilab*

```
      !--error     8
inconsistent addition

-->E = A*C           // C est une matrice (3,4) dont les elements sont des 1
 E  =

!   3.     3.     3.     3.  !
!   14.    14.    14.    14. !
!   39.    39.    39.    39. !

--> C*A   // produit de matrice impossible (3,4)x(3,3) : que dit Scilab ?
     !--error    10
inconsistent multiplication

--> At = A'  // la transposee s'obtient en postfixant la matrice par une apostrophe
 At  =

!   1.    2.    3.  !
!   1.    4.    9.  !
!   1.    8.    27. !

--> Ac = A + %i*eye(3,3)  // je forme ici une matrice a coef. complexes
 Ac  =
```



```
!   1. + i      1.          1.        !
!   2.          4. + i      8.        !
!   3.          9.          27. + i   !

--> Ac_adj = Ac' // dans le cas complexe ' donne l'adjointe (transposee conjuguee)
 Ac_adj  =

!   1. - i      2.          3.        !
!   1.          4. - i      9.        !
!   1.          8.          27. - i   !

-->x = linspace(0,1,5)' // je forme un vecteur colonne
 x  =

!   0.   !
!   0.25 !
!   0.5  !
!   0.75 !
!   1.   !

-->y = (1:5)' // un autre vecteur colonne
 y  =

!   1. !
!   2. !
!   3. !
!   4. !
!   5. !

-->p = y'*x   // produit scalaire (x | y)
 p  =

    10.

-->Pext = y*x' // on obtient une matrice (5,5) ((5,1)x(1,5)) de rang 1 : pourquoi ?
 Pext  =

!   0.    0.25    0.5    0.75    1. !
!   0.    0.5     1.     1.5     2. !
!   0.    0.75    1.5    2.25    3. !
!   0.    1.      2.     3.      4. !
!   0.    1.25    2.5    3.75    5. !

--> Pext / 0.25   // on peut diviser une matrice par un scalaire
 ans  =

!   0.    1.    2.     3.     4.  !
!   0.    2.    4.     6.     8.  !
!   0.    3.    6.     9.     12. !
!   0.    4.    8.     12.    16. !
!   0.    5.    10.    15.    20. !

--> A^2    // elevation a la puissance d'une matrice
 ans  =
```

```
!   6.      14.     36.  !
!   34.     90.     250. !
!   102.    282.    804. !

--> [0 1 0] * ans   // on peut reutiliser la variable ans (qui contient
-->                 // le dernier resultat non affecte a une variable)
 ans  =

!   34.    90.    250. !

--> Pext*x - y + rand(5,2)*rand(2,5)*ones(x) + triu(Pext)*tril(Pext)*y;
--> // taper ans pour voir le resultat
```

Une autre caractéristique très interessante est que les fonctions usuelles (voir table 2.1) s'appliquent aussi aux matrices « élément par élément » : si `f` désigne une telle fonction `f(A)` est la matrice $[f(a_{ij})]$. Quelques exemples :

```
-->sqrt(A)
 ans  =

!   1.          1.    1.        !
!   1.4142136   2.    2.8284271 !
!   1.7320508   3.    5.1961524 !

-->exp(A)
 ans  =

!   2.7182818    2.7182818    2.7182818 !
!   7.3890561    54.59815     2980.958  !
!   20.085537    8103.0839    5.320D+11 !
```

Rmq : pour les fonctions qui ont un sens pour les matrices (différent de celui qui consiste à l'appliquer sur chaque élément...), par exemple l'exponentielle, le nom de la fonction est suivi par `m`. Ainsi pour obtenir l'exponentielle de A, on rentre la commande :

```
-->expm(A)
 ans  =

   1.0D+11 *

!   0.5247379    1.442794     4.1005925 !
!   3.6104422    9.9270989    28.213997 !
!   11.576923    31.831354    90.468498 !
```

2.3.2 Opérations « élément par élément »

Pour multiplier et diviser deux matrices A et B de même dimensions en appliquant ces opérations « élément par élément » on utilise les opérateurs `.*` et `./` : `A.*B` est la matrice $[a_{ij}b_{ij}]$ et `A./B` $[a_{ij}/b_{ij}]$. De même, on peut élever à la puissance chaque coefficient en utilisant l'opérateur postfixé `.^` : `A.^p` permet d'obtenir la matrice $[a_{ij}^p]$. Essayer par exemple :

```
-->A./A
ans =

!  1.  1.  1. !
!  1.  1.  1. !
!  1.  1.  1. !
```

Remarques :

- tant que `A` n'est pas une matrice carrée, `A^n` va fonctionner au sens « élément par élément », je conseille néanmoins d'utiliser `A.^n` car cette écriture est plus claire pour exprimer cette intention ;
- si `s` est un scalaire et `A` une matrice, `s.^A` donnera la matrice $[s^{a_{ij}}]$.

2.3.3 Résoudre un système linéaire

Pour résoudre un système linéaire dont la matrice est carrée, Scilab utilise une factorisation LU avec pivot partiel suivie de la résolution des deux systèmes triangulaires. Cependant ceci est rendu transparent pour l'utilisateur par l'intermédiaire de l'opérateur `\`, essayer :

```
-->b=(1:3)'      //je cree un second membre b
 b  =

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

-->x=A\b         // on resout Ax=b
 x  =

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

-->A*x - b    // je verifie le resultat en calculant le vecteur residu
 ans  =

!   0. !
!   0. !
!   0. !
```

Pour se souvenir de cette instruction, il faut avoir en tête le système initial $Ax = y$ puis faire comme si on multipliait à gauche par A^{-1}, (ce que l'on symbolise par une division à gauche par A) d'où la syntaxe utilisée par Scilab. Ici on a obtenu un résultat exact mais en général, il y a des erreurs d'arrondi dues à l'arithmétique flottante :

```
-->R = rand(100,100);  // mettre le ; pour ne pas submerger l'ecran de chiffres

-->y = rand(100,1);    // meme remarque

-->x=R\y;              // resolution de Rx=y

-->norm(R*x-y)    // norm permet de calculer la norme de vecteurs (et aussi de matrices)
                  // (par defaut la norme 2 (euclidienne ou hermitienne))
 ans  =

    1.134D-13
```

Rmq : vous n'obtiendrez pas forcément ce résultat si vous n'avez pas joué avec la fonction `rand` exactement comme moi... Lorsque la résolution d'un système linéaire semble douteuse Scilab renvoie quelques informations permettant de prévenir l'utilisateur (cf compléments sur la résolution des systèmes linéaires).

2.3.4 Référencer, extraire, concaténer matrices et vecteurs

Les coefficients d'une matrice peuvent être référencés avec leur(s) indice(s) précisés entre parenthèses, (). Par exemple :

```
-->A33=A(3,3)
 A33  =

    27.

-->x_30 = x(30,1)
 x_30  =

  - 1.2935412

-->x(1,30)
        !--error    21
invalid index


-->x(30)
 ans  =

  - 1.2935412
```

Rmq : si la matrice est un vecteur colonne, on peut se contenter de référencer un élément en précisant uniquement son indice de ligne, et inversement pour un vecteur ligne.

Un des avantages d'un langage comme Scilab est que l'on peut extraire des sous-matrices tout aussi aisément. Quelques exemples simples pour commencer :

```
-->A(:,2)   // pour extraire la 2 eme colonne
 ans  =

!   1. !
!   4. !
!   9. !

-->A(3,:)   // la 3 eme ligne
 ans  =

!   3.    9.    27. !

-->A(1:2,1:2)  // la sous matrice principale d'ordre 2
 ans  =

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

Passons maintenant à la syntaxe générale : si A est une matrice de taille (n,m), et si $v1 = (i_1,i_2,\ldots,i_p)$ et $v2 = (j_1,j_2,\ldots,j_q)$ sont deux vecteurs (ligne ou colonne peut importe) d'indices dont les valeurs sont telles que $1 \leq i_k \leq n$ et $1 \leq j_k \leq m$ alors `A(v1,v2)` est la matrice (de dimension (p,q)) formée par l'intersection des lignes $i_1,i_2,\ldots,i_p$ et des colonnes $j_1,j_2,\ldots,j_q$. Exemples :

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

!   1.    1.  !
!   9.    27. !

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

!   9.    3. !
!   1.    1. !
```

Dans la pratique on utilise généralement des extractions plus simples, comme celle d'un bloc contigu ou bien d'une (ou plusieurs) colonne(s) ou ligne(s). Dans ce cas, on utilise l'expression `i_debut:incr:i_fin` pour générer les vecteurs d'indices, ainsi que le caractère `:` pour désigner toute l'étendue dans la dimension adéquate (cf premiers exemples). Ainsi pour obtenir la sous-matrice formée de la première et troisième ligne :

```
-->A(1:2:3,:)    // ou encore A([1 3],:)
 ans  =

!   1.    1.    1.  !
!   3.    9.    27. !
```

Passons maintenant à la concaténation de matrices qui est l'opération permettant d'assembler (en les juxtaposant) plusieurs matrices, pour en obtenir une autre. Voici un exemple : on considère la matrice suivante, avec un découpage par blocs :

$$A = \left(\begin{array}{c|ccc} 1 & 2 & 3 & 4 \\ \hline 1 & 4 & 9 & 16 \\ 1 & 8 & 27 & 64 \\ 1 & 16 & 81 & 256 \end{array}\right) = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}.$$

On va d'abord définir les sous-matrices $A_{11},A_{12},A_{21},A_{22}$:

```
-->A11=1;

-->A12=[2 3 4];

-->A21=[1;1;1];

-->A22=[4 9 16;8 27 64;16 81 256];
```

enfin on obtient A par concaténation de ces 4 blocs :

```
-->A=[A11 A12; A21 A22]
A =

!  1.  2.  3.   4.   !
!  1.  4.  9.   16.  !
!  1.  8.  27.  64.  !
!  1.  16. 81.  256. !
```

pour la syntaxe tout se passe comme si nos sous-matrices étaient de simples scalaires (il faut bien sûr une certaine compatibilité entre le nombre de lignes et de colonnes des différents blocs...).

Il existe une syntaxe particulière pour détruire un ensemble de lignes ou de colonnes d'une matrice : si $v = (k_1,k_2,\ldots,k_p)$ est un vecteur d'indices repérant des numéros de lignes ou de colonnes d'une matrice M alors `M(v,:) = []` détruit les lignes $k_1,k_2,\ldots,k_p$ de M et `M(:,v) = []` détruit les colonnes $k_1,k_2,\ldots,k_p$. Enfin, pour un vecteur (ligne ou colonne) u, `u(v) = []` détruit les entrées correspondantes.

2.4 Sauvegarder des données et les lire

Lorsque l'on a défini une matrice (ou un vecteur) de réels dans l'environnement, il est possible de la sauvegarder (en ascii) dans un fichier, voici un exemple:

```
-->A     // juste pour revoir la tete de A

 A  =

!   1.    2.     3.     4.   !
!   1.    4.     9.     16.  !
!   1.    8.     27.    64.  !
!   1.    16.    81.    256. !

-->write('mat.dat',A)     // mat.dat est le nom du fichier
```

Et si une matrice est stockée dans un fichier de données (en ascii), on peut la lire complètement ou partiellement avec l'instruction `read`. En reprenant l'exemple précédent :

```
-->Anew=read('mat.dat',2,2)
Anew =

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

-->Anew=read('mat.dat',4,2)
Anew =

!  1.   2.  !
!  1.   4.  !
!  1.   8.  !
!  1.   16. !

-->Anew=read('mat.dat',4,4)
Anew =

!  1.   2.  3.    4.    !
!  1.   4.  9.    16.   !
!  1.   8.  27.   64.   !
!  1.   16. 81.   256.  !
```

Les arguments de cette commande sont respectivement le nom du fichier, le nombre de lignes et le nombre de colonnes (que l'on désire lire) de la matrice. Si on ne connait pas le nombre exact de lignes, on peut le remplacer par -1 (et ainsi la lecture se fait jusqu'à la fin du fichier). Ces fonctions, conjuguées avec la fonction `file`, admettent des possibilités de lecture / écriture plus souples (cf Programmation). Pour les habitués du langage C, il existe aussi une émulation des fonctions `fscanf` et `fprintf`.

On peut aussi sauvegarder en binaire, puis recharger, toutes ou partie des variables que l'on a définies avec les instructions `load`, et `save` :

- `save('nomfich')` sauvegarde toutes les variables dans le fichier `nomfich`
- `save('nomfich',x1,x2,...)` sauvegarde dans `nomfich` uniquement les variables x1, x2, ...
- `load('nomfich')` charge toutes les variables qui avaient été sauvegardées dans `nomfich`
- `load('nomfich','x1','x2',...)` charge uniquement les variables de nom x1, x2, ... parmi les variables sauvegardées dans `nomfich`

2.5 Information sur l'espace de travail (*)

Il suffit de rentrer la commande :

```
-->who
your variables are...

 Anew       A          A22        A21        A12        A11        x_30       A33        x
 y          R          b          Pext       p          Ac_adj     Ac         At         E
 D          cosh       ind        xx         i          linspace   M          U          O
 zeros      C          B          I          Y          c          T          startup    ierr
 scicos_pal            home       PWD        TMPDIR     percentlib            fraclablib
 soundlib   xdesslib   utillib    tdcslib    siglib     s2flib     roblib     optlib     metalib
 elemlib    commlib    polylib    autolib    armalib    alglib     mtlblib    SCI        %F
 %T         %z         %s         %nan       %inf       old        newstacksize          $
 %t         %f         %eps       %io        %i         %e         %pi
 using      14875 elements  out of    1000000.
          and         75 variables out of       1023
```

et l'on voit apparaître :

- les variables que l'on a rentrées sous l'environnement : `Anew, A, A22, A21`, ..., `b` dans l'ordre inverse de leur création. En fait la première variable créée était la matrice A mais nous avons « augmenté » ses dimensions (de (3,3) à (4,4)) lors de l'exemple concernant la concaténation de matrice. Dans un tel cas, la variable initiale est détruite pour être recréée avec ses nouvelles dimensions. Ceci est un point important dont nous reparlerons lors de la programmation en Scilab ;
- les noms des bibliothèques de Scilab (qui se terminent par lib) et un nom de fonction : cosh. En fait, les fonctions (celles écrites en langage Scilab) et les bibliothèques sont considérées comme des variables par Scilab ; *Rmq :* les « procédures » Scilab programmées en Fortran 77 et en C sont appelées « primitives Scilab » et ne sont pas considérées comme des variables Scilab ; dans la suite du document j'utilise parfois abusivement le terme « primitives » pour désigner des fonctions Scilab (programmées en langage Scilab) qui sont proposées par l'environnement standard ;
- des constantes prédéfinies comme π,e, l'unité imaginaire i, la précision machine eps et les deux autres constantes classiques de l'arithmétique flottante (nan not a number) et inf (pour ∞) ; ces variables dont le nom débute nécessairement par `%` ne peuvent pas être détruites ;
- une variable importante `newstacksize` qui correspond à la taille (par défaut) de la pile (c-à-d de la mémoire disponible).
- ensuite Scilab indique le nombre de mots de 8 octets utilisés ainsi que la mémoire totale disponible (taille de la pile) puis le nombre de variables utilisées ainsi que le nombre maximum autorisé.

On peut changer la taille de la pile à l'aide de la commande `stacksize(nbmots)` où `nbmots` désigne la nouvelle taille désirée, et la même commande sans argument `stacksize()` permet d'obtenir la taille de la pile ainsi que le nombre maximum autorisé de variables.

Enfin si l'on veut supprimer une variable `v1` de l'environnement, (et donc regagner de la place mémoire) on utilise la commande : `clear v1`. La commande `clear` utilisée seule détruit toutes vos variables et si vous voulez simplement détruire les variables `v1, v2, v3`, il faut utiliser `clear v1 v2 v3`.

2.6 Utilisation de l'aide en ligne

Elle s'obtient en cliquant sur le bouton `Help` de la fenêtre Scilab... Apparaît alors une nouvelle fenêtre (intitulée *Scilab Help Panel*) qui se décompose en trois parties (plus le bouton `done` qui détruit cette fenêtre). La partie du milieu correspond à un classement de toutes les fonctions en un certain nombre de rubriques (`Scilab Programming, Graphic Library, Utilities and Elementary functions,...`) alors que la première partie donne la liste de toutes les fonctions [2] de la rubrique qui apparaît en inverse vidéo dans la deuxième partie... Pour changer de rubrique il suffit de cliquer une seule fois sur la rubrique désirée. Pour obtenir le détail d'une fonction il suffit de cliquer (toujours une seule fois) sur son intitulé :

2. pour chaque fonction apparaît son nom suivi d'une brève description de quelques mots

apparaît alors une autre fenêtre donnant ces détails. Pour fermer cette dernière fenêtre, il faut cliquer sur le bouton `close window` (mais garder la, tant que vous en avez besoin). Lorsque l'on ne sait pas où chercher on peut écrire un mot clé dans la troisième partie intitulée `Apropos` puis appuyer sur la touche `Return` pour lancer la recherche. Si la recherche échoue, vous avez le message « `no info for topics ... back to chapter one` » qui apparaît (signifiant que la première partie de la fenêtre d'aide affiche les fonctions de la rubrique 1 (Scilab Programming)). En cas de succès, la première partie affiche toutes les fonctions qui contiennent ce mot clé dans leur intitulé (c-à-d dans le nom de la fonction ou dans sa brève description). Par exemple, la recherche sur le mot clé `int` renvoie énormément de fonctions car la chaîne de caractères `int` est présente dans les mots interrupt, interface, printing, points, etc. Si je fais suivre `int` par un caractère blanc, on sélectionne déjà moins de fonctions et l'on voit alors clairement apparaître cette fonction dans la liste sélectionnée.

D´autre part, si vous voulez la page d'aide d´une primitive Scilab dont vous connaissez le nom (supposons que ce soit `fonc`), vous pouvez simplement rentrer la commande `help fonc` dans la fenêtre principale (après l´invite `-->`) et la page s´affiche alors directement (inutile alors de passer par le «*Scilab Help Panel* »).

2.7 Visualiser un graphe simple

Supposons que l'on veuille visualiser la fonction $y = e^{-x}\sin(4x)$ pour $x \in [0,2\pi]$. On peut tout d'abord créer un maillage de l'intervalle par la fonction `linspace`:

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

puis, calculer les valeurs de la fonction pour chaque composante du maillage, ce qui, grâce aux instructions « vectorielles » ne nécessite aucune boucle:

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

et enfin :

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

où les trois dernières chaînes de caractères (respectivement une légende pour les abscisses, une pour les ordonnées et un titre) sont facultatives. L'instruction permet de tracer une courbe passant par les points dont les coordonnées sont données dans les vecteurs `x` pour les abscisses et `y` pour les ordonnées. Comme les points sont reliés par des segments de droites, le tracé sera d'autant plus fidèle que les points seront nombreux.

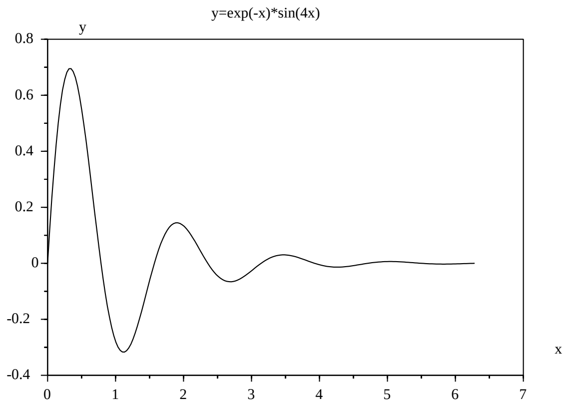


FIG. 2.1 – *Un graphe simple*

Rmq : cette instruction est limitée car vous ne pouvez afficher qu'une seule courbe. Dans le chapitre sur les graphiques, on apprendra à utiliser `plot2d` qui est beaucoup plus puissante.

2.8 Écrire et exécuter un script

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

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

Une méthode plus conviviale est de sélectionner l'item `File Operations` proposé par le menu obtenu en appuyant sur le bouton `File` de la fenêtre Scilab. On obtient alors un menu qui permet de sélectionner son fichier (éventuellement en changeant le répertoire courant) et il ne reste plus qu'à cliquer sur le bouton `Exec`. Comme exemple de script, reprenons le tracé de la fonction $e^{-x}\sin(4x)$ en proposant de plus le choix de l'intervalle de visualisation $[a,b]$ ainsi que sa discrétisation. J'écris donc dans un fichier intitulé par exemple `script1.sce` les instructions Scilab suivantes :

```
// mon premier script Scilab

a = input(" Rentrer la valeur de a : ");
b = input(" Rentrer la valeur de b : ");
n = input(" Nb d''intervalles n : ");

// calcul des abscisses
x = linspace(a,b,n+1);

// calcul des ordonnees
y = exp(-x).*sin(4*x);

// un petit dessin
plot(x,y,'x','y','y=exp(-x)*sin(4x)')
```

Rmq : 1/ pour s'y retrouver dans vos fichiers Scilab, il est recommandé de suffixer le nom des fichiers scripts par la terminaison `.sce` (alors qu'un fichier contenant des fonctions sera suffixé en `.sci`); 2/ l'éditeur `emacs` peut être muni d'un mode d'édition (à récupérer sur la home page Scilab) qui facilite l'écriture de programmes Scilab; 3/ un script sert souvent de programme principal d'une application écrite en Scilab.

2.9 Compléments divers

2.9.1 Quelques raccourcis d'écriture dans les expressions matricielles

Nous avons vu précédemment que la multiplication d'une matrice par un scalaire est reconnue par Scilab, ce qui est naturel (de même la division d'une matrice par un scalaire). Par contre Scilab utilise des raccourcis moins évidents comme l'addition d'un scalaire et d'une matrice. L'expression `M + s` où `M` est une matrice et `s` un scalaire est un raccourci pour :

```
M + s*ones(M)
```

c'est à dire que le scalaire est ajouté à tous les éléments de la matrice.

Autre raccourci : dans une expression du type `A./B` (qui correspond normalement à la division « élément par élément » de deux matrices de même dimension), si `A` est un scalaire alors l'expression est un raccourci pour :

```
A*ones(B)./B
```

on obtient donc la matrice $[a/b_{ij}]$. Ces raccourcis permettent une écriture plus synthétique dans de nombreux cas (cf exercices). Par exemple, si f est une fonction définie dans l'environnement, x un vecteur et s une variable scalaire alors :

```
s./f(x)
```

est un vecteur de même taille que x dont la i$^{\text{ème}}$composante est égale à $s/f(x_i)$. Ainsi pour calculer le vecteur de composante $1/f(x_i)$, il semble que l'on puisse utiliser :

```
1./f(x)
```

mais comme `1.` est syntaxiquement égal à `1`, le résultat n'est pas celui escompté. La bonne manière d'obtenir ce que l'on cherche est d'entourer le nombre avec des parenthèses ou de rajouter un blanc entre le nombre et le point :

```
(1)./f(x)       // ou encore 1 ./f(x)
```

2.9.2 Remarques diverses sur la résolution de systèmes linéaires (*)

1. Lorsque l'on a plusieurs seconds membres, on peut procéder de la façon suivante :

```
-->y1 = [1;0;0;0]; y2 = [1;2;3;4];  // voici 2 seconds membres (on peut mettre
-->                                 // plusieurs instructions sur une seule ligne)
-->X=A\[y1,y2]     // concaténation de y1 et y2
 X  =
!   4.         - 0.8333333 !
! - 3.           1.5       !
!   1.3333333  - 0.5       !
! - 0.25         0.0833333 !
```

la première colonne de la matrice X est la solution du système linéaire $Ax^1 = y^1$, alors que la deuxième correspond à la solution de $Ax^2 = y^2$.

2. Nous avons vu précédemment que si A est une matrice carrée (n,n) et b un vecteur colonne à n composantes (donc une matrice (n,1)) alors :

```
x = A\b
```

nous donne la solution du système linéaire $Ax = b$. Si la matrice A est détectée comme étant singulière, Scilab renvoie un message d'erreur. Par exemple :

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

-->b=[1;1];

-->A\b
    !--error    19
singular matrix
```

Cependant si la matrice A est considérée comme mal conditionnée (ou éventuellement mal équilibrée) une réponse est fournie mais elle est accompagnée d'un message de mise en garde avec une estimation de l'inverse du conditionnement ($cond(A) = ||A||\ ||A^{-1}||$):

```
-->A=[1 2;1 2+3*%eps];
-->A\b
 warning
 matrix is close to singular or badly scaled.
 results may be inaccurate. rcond =   7.4015D-17

 ans  =

!   1. !
!   0. !
```

Par contre si votre matrice n'est pas carrée, tout en ayant le même nombre de lignes que le second membre, Scilab va vous renvoyer une solution (un vecteur colonne de dimension le nombre de colonnes de A) sans s'émouvoir (sans afficher en général de message d'erreur). En effet, si

dans ce cas l'équation $Ax = b$ n'a généralement pas une solution unique[3], on peut toujours sélectionner un vecteur unique x qui vérifie certaines propriétés (x de norme minimale et solution de min $||Ax - b||$). Dans ce cas, la résolution est confiée à d'autres algorithmes qui vont permettre d'obtenir (éventuellement) cette pseudo-solution[4]. L'inconvénient est que si vous avez fait une erreur dans la définition de votre matrice (par exemple vous avez défini une colonne supplémentaire, et votre matrice est de taille (n,n+1)) vous risquez de ne pas vous en apercevoir immédiatement. En reprenant l'exemple précédent :

```
-->A(2,3)=1        // étourderie
A =

!  1.   2.   0. !
!  1.   2.   1. !

-->A\b
 ans  =

!   0.         !
!   0.5        !
! - 3.140D-16 !

-->A*ans - b
 ans  =

   1.0D-15 *

! - 0.1110223 !
! - 0.1110223 !
```

En dehors de vous mettre en garde sur les conséquences de ce type d'étourderie, l'exemple est instructif sur les points suivants :

- `x = A\y` permet donc de résoudre aussi un problème de moindres carrés (lorsque la matrice n´est pas de rang maximum, il vaut mieux utiliser `x = pinv(A)*b`, la pseudo-inverse étant calculée via la décomposition en valeurs singulières de A (cette décomposition peut s´obtenir avec la fonction `svd`) ;
- l'instruction `A(2,3)=1` (l' erreur d'étourderie…) est en fait un raccourci pour :

  ```
  A = [A,[0;1]]
  ```

  c'est à dire que Scilab détecte que vous voulez compléter la matrice A (par une troisième colonne) mais il lui manque un élément. Dans ce cas, il complète par des zéros.
- l'élément en position (2,2) est normalement égal à $2 + 3\,\epsilon_m$ aux erreurs d'arrondi numérique près. Or le epsilon machine (ϵ_m) peut être défini comme le plus grand nombre pour lequel $1 \oplus \epsilon_m = 1$ en arithmétique flottante[5]. Par conséquent on devrait avoir $2 \oplus 3\epsilon_m > 2$ alors que la fenêtre affiche 2. Ceci vient du format utilisé par défaut mais on peut le modifier par l'instruction `format` :

  ```
  -->format('v',19)

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

3. soit (m,n) les dimensions de A (telles que $n \neq m$), on a une solution unique si et seulement si $m > n$, $KerA = \{0\}$ et enfin $b \in ImA$ cette dernière condition étant exceptionnelle si b est pris au hasard dans $\mathbb{K}^m$; dans tous les autres cas, on a soit aucune solution, soit une infinité de solutions

4. dans les cas difficiles, c-à-d lorsque la matrice n'est pas de rang maximum ($rg(A) < min(n,m)$ où n et m sont les 2 dimensions) il vaut mieux calculer cette solution en passant par la pseudo-inverse de A (`x = pinv(A)*b`).

5. en fait tout nombre réel x tel que $m \le |x| \le M$ peut être codé par un nombre flottant $fl(x)$ avec : $|x - fl(x)| \le \epsilon_m |x|$ où m et M sont respectivement le plus petit et le plus grand nombre positif codable en virgule flottante normalisée

```
2.0000000000000009
```
alors que l'affichage par défaut correspond à `format('v',10)` (voir le `Help` pour la signification des arguments).

– Lorque l'on a calculé la « solution » de $Ax = b$ on ne l'a pas affecté à une variable particulière et Scilab s'est donc servi de **ans** que j'ai ensuite utilisé pour calculer le résidu $Ax - b$.

3. Avec Scilab on peut aussi directement résoudre un système linéaire du type $xA = b$ où x et b sont des vecteurs lignes et A une matrice carrée (en transposant on se ramène à un système linéaire classique $A^\top x^\top = b^\top$), il suffit de faire comme si l'on multipliait à droite par A^{-1} (en symbolisant cette opération par une division à droite par A) :
```
x = b/A
```
et de même que précédemment, si A est une matrice rectangulaire (dont le nombre de colonnes est égal à celui de b) Scilab renvoie une solution, il faut donc, là aussi, faire attention.

2.9.3 Quelques primitives matricielles supplémentaires (*)

Somme, produit des coefficients d' une matrice, matrice vide

Pour faire la somme des coefficients d'une matrice, on utilise `sum` :
```
-->sum(1:6)   // 1:6 = [1 2 3 4 5 6] : on doit donc obtenir 6*7/2 = 21  !!!!!
 ans  =

    21.
```
Cette fonction admet un argument supplémentaire pour effectuer la somme selon les lignes ou les colonnes :
```
-->B = [1 2 3; 4 5 6]
 B  =

!   1.    2.    3. !
!   4.    5.    6. !

-->sum(B,"row")  // effectue la somme de chaque colonne -> on obtient une ligne
 ans  =

!   5.    7.    9. !

-->sum(B,"col") // effectue la somme de chaque ligne -> on obtient une colonne
 ans  =

!   6.   !
!   15.  !
```
Il existe un objet très pratique « la matrice vide » que l'on définit de la façon suivante :
```
-->C =[]
 C  =

     []
```
La matrice vide intéragit avec d'autres matrices avec les règles suivantes : `[] + A = A` et `[]*A = []`. Si on applique maintenant la fonction `sum` sur cette matrice vide, on obtient le résultat naturel :
```
-->sum([])
 ans  =

    0.
```


identique à la convention utilisée en mathématique pour les sommations :

$$S = \sum_{i \in E} u_i = \sum_{i=1}^{n} u_i \quad \text{si } E = \{1,2,\dots,n\}$$

lorsque l'ensemble E est vide, on impose en effet par convention $S = 0$.

De manière analogue, pour effectuer le produit des éléments d'une matrice, on dispose de la fonction `prod` :
```
-->prod(1:5)     // en doit obtenir 5! = 120
 ans  =

    120.

-->prod(B,"row")  // taper B pour revoir cette matrice...
 ans  =

!   4.    10.    18. !

-->prod(B,"col")
 ans  =

!   6.   !
!   120. !

-->prod(B)
 ans  =

    720.

-->prod([])
 ans  =

    1.
```
et l'on obtient toujours la convention usuelle rencontrée en mathématique :

$$\prod_{i \in E} u_i = 1, \ \text{si } E = \emptyset.$$

Remodeler une matrice

La fonction `matrix` permet de remodeler une matrice en donnant de nouvelles dimensions (mais avec le même nombre de coefficients en tout) .
```
-->B_new = matrix(B,3,2)   // retaper encore B...
 B_new  =

!   1.    5. !
!   4.    3. !
!   2.    6. !
```
Elle travaille en ordonnant les coefficients colonne par colonne. Une de ses utilisations est de transformer un vecteur ligne en vecteur colonne et inversement. Signalons encore un raccourci qui permet

de transformer une matrice A (vecteurs ligne et colonne compris) en un vecteur colonne v : v = A(:), exemple :

```
-->A = rand(2,2)
 A  =

!   0.8782165    0.5608486 !
!   0.0683740    0.6623569 !

-->v=A(:)
 v  =

!   0.8782165 !
!   0.0683740 !
!   0.5608486 !
!   0.6623569 !
```

Vecteurs avec espacement logarithmique

Parfois on a besoin d'un vecteur avec une incrémentation logarithmique pour les composantes (c-à-d tel que le rapport entre deux composantes successives soit constant : $x_{i+1}/x_i = Cte$) : on peut utiliser dans ce cas la fonction `logspace` : `logspace(a,b,n)` : permet d'obtenir un tel vecteur avec n composantes, dont la première et la dernière sont respectivement 10^a et 10^b, exemple :

```
-->logspace(-2,5,8)
 ans  =

!   0.01    0.1    1.    10.    100.    1000.    10000.    100000. !
```

Valeurs et vecteurs propres

La fonction `spec` permet de calculer les valeurs propres d'une matrice (carrée !) :

```
-->A = rand(5,5)
 A  =

!   0.2113249    0.6283918    0.5608486    0.2320748    0.3076091 !
!   0.7560439    0.8497452    0.6623569    0.2312237    0.9329616 !
!   0.0002211    0.6857310    0.7263507    0.2164633    0.2146008 !
!   0.3303271    0.8782165    0.1985144    0.8833888    0.312642  !
!   0.6653811    0.0683740    0.5442573    0.6525135    0.3616361 !

-->spec(A)
 ans  =

!   2.4777836              !
! - 0.0245759 + 0.5208514i !
! - 0.0245759 - 0.5208514i !
!   0.0696540              !
!   0.5341598              !
```

et renvoie le résultat sous forme d'un vecteur colonne (Scilab utilise la méthode QR qui consiste à obtenir itérativement une décomposition de *Schur* de la matrice). Les vecteurs propres peuvent s'obtenir avec `bdiag`. Pour un problème de valeurs propres généralisé, vous pouvez utiliser la fonction `gspec`.

2.9.4 Les fonctions `size` et `length`

`size` permet de récupérer les deux dimensions (nombre de lignes puis de colonnes) d'une matrice :

```
-->[nl,nc]=size(B)   // B est la matrice (2,3) de l'exemple precedent
 nc  =

    3.
 nl  =

    2.

-->x=5:-1:1
 x  =

!   5.    4.    3.    2.    1. !

-->size(x)
 ans  =

!   1.    5. !
```

alors que `length` fournit le nombre d'éléments d'une matrice (réelle ou complexe). Ainsi pour un vecteur ligne ou colonne, on obtient directement son nombre de composantes :

```
-->length(x)
 ans  =

    5.

-->length(B)
 ans  =

    6.
```

En fait ces deux primitives seront surtout utiles à l'intérieur de fonctions pour récupérer les tailles des matrices et vecteurs, ce qui évitera de les faire passer comme arguments. Noter aussi que `size(A,'r')` (ou `size(A,1)`) et `size(A,'c')` (ou `size(A,2)`) permettent d'obtenir le nombre de lignes (rows) et de colonnes (columns) de la matrice A.

2.10 Exercices

1. Définir la matrice d'ordre n suivante (voir le détail de la fonction `diag` à l'aide du `Help`) :

$$A = \begin{pmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & \ddots & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & -1 \\ 0 & \cdots & 0 & -1 & 2 \end{pmatrix}$$

2. Soit A une matrice carrée ; que vaut `diag(diag(A))` ?
3. Les fonctions `tril` (resp. `triu`) permet d'extraire la partie triangulaire inférieure (resp. supérieure) d'une matrice. Définir une matrice carrée A quelconque (par exemple avec `rand`) et construire en une seule instruction, une matrice triangulaire inférieure T telle que $t_{ij} = a_{ij}$ pour $i > j$ (les parties strictement triangulaires inférieures de A et T sont égales) et telle que $t_{ii} = 1$ (T est à diagonale unité).

4. Soit `X` une matrice (ou un vecteur…) que l'on a définie dans l'environnement. Écrire l'instruction qui permet de calculer la matrice `Y` (de même taille que `X`) dont l'élément en position (i,j) est égal à $f(X_{ij})$ dans les cas suivants :
   (a) $f(x) = 2x^2 - 3x + 1$
   (b) $f(x) = |2x^2 - 3x + 1|$
   (c) $f(x) = (x-1)(x+4)$
   (d) $f(x) = \frac{1}{1+x^2}$
5. Tracer le graphe de la fonction $f(x) = \frac{\sin x}{x}$ pour $x \in [0,4\pi]$ (écrire un script).
6. Quelle est la taille maximum d'une matrice carrée que l'on peut rentrer dans l'environnement de Scilab (en supposant que c'est la seule variable que l'on va créer en plus de la variable `ans` qui pourra servir à faire le calcul de la dimension de cette matrice) ?
   Aide : on pourra d'abord taper who pour connaître le nombre de mots libres et on pourra s'apercevoir que la création d'une variable consomme 2 mots dans la pile (en plus des mots utilisés pour le contenu).

Chapitre 3

La programmation en Scilab

Scilab , en dehors des primitives toutes prêtes (qui permettent avec les instructions matricielles, une programmation très synthétique proche du langage mathématique, du moins matriciel) dispose d'un langage de programmation simple mais assez complet. La différence essentielle par rapport aux langages habituels (C, C++, Pascal, ...) est que les variables ne sont pas déclarées : lors des manipulations précédentes nous n'avons à aucun moment précisé la taille des matrices, ni même leurs types (réelles, complexes, ...). C'est l'interprèteur de Scilab qui, lorsqu'il rencontre un nouvel objet, se charge de ce travail. Cette caractéristique est souvent déconsidérée (avec raison, cf l'ancien fortran) d'un point de vue génie logiciel car cela peut conduire à des erreurs difficilement repérables. Dans le cas de langage comme Scilab (ou MATLAB) cela ne pose pas trop de problèmes, car la notation vectorielle/matricielle et les primitives toutes prêtes, permettent de restreindre considérablement le nombre de variables et de lignes d'un programme (par exemple, les instructions matricielles permettent d'éviter un maximum de boucles). Un autre avantage de ce type de langage, est de disposer d'instructions graphiques (ce qui évite d'avoir à jongler avec un programme ou une bibliothèque graphique) et d'être interprêté (ce qui permet de chasser les erreurs assez facilement, sans l'aide d'un débogueur). Par contre l'inconvénient est qu'un programme écrit en Scilab est plus lent (voire beaucoup plus lent) que si vous l'aviez écrit en C (mais le programme en C a demandé 50 fois plus de temps d'écriture et de mise au point...). D'autre part, pour les applications où le nombre de données est vraiment important (par exemple des matrices 1000×1000) il vaut mieux revenir à un langage compilé. Un point important concernant la rapidité d'exécution est qu'il faut programmer en utilisant au maximum les primitives disponibles et les instructions matricielles (c'est à dire : limiter le nombre de boucles au maximum)[1]. En effet, dans ce cas, Scilab appelle alors une routine (fortran) compilée, et donc son interprèteur travaille moins… Pour bénéficier du meilleur des deux mondes (c'est dire de la rapidité d'un langage compilé et du confort d'un environnement comme celui de Scilab), vous avez des facilités pour lier à Scilab des sous-programmes fortran (77) ou C.

3.1 Les boucles

Il existe deux types de boucles : la boucle `for` et la boucle `while`.

3.1.1 La boucle `for`

La boucle `for` itère sur les composantes d'un vecteur ligne :

```
-->v=[1 -1 1 -1]
-->y=0; for k=v, y=y+k, end
```

Le nombre d'itérations est donné par le nombre de composantes du vecteur ligne[2], et à la i$^{\text{ème}}$ itération, la valeur de `k` est égale à `v(i)`. Pour que les boucles de Scilab ressemblent à des boucles du type :

pour $i := i_{deb}$ **à** i_{fin} **par pas de** i_{step} **faire :**
  suite d'instructions
fin pour

1. voir la section « Quelques remarques sur la rapidité »
2. si le vecteur est une matrice vide alors aucune itération n'a lieu

il suffit d'utiliser comme vecteur `i_deb:i_step:i_fin` , et lorsque l'incrément `i_step` est égal à 1 nous avons vu qu'il peut peut être omis. La boucle précédente peut alors être écrite plus naturellement de la façon suivante :

```
-->y=0; for i=1:4, y=y+v(i), end
```

Quelques remarques :

- une boucle peut aussi itérer sur une matrice. Le nombre d'itérations est égal au nombre de colonnes de la matrice et la variable de la boucle à la $i^{\text{ème}}$ itération est égale à la $i^{\text{ème}}$ colonne de la matrice. Voici un exemple :

  ```
  -->A=rand(3,3);y=zeros(3,1); for k=A, y = y + k, end
  ```

- la syntaxe précise est la suivante :

  ```
  for variable = matrice, suite d'instructions, end
  ```

  où les instructions sont séparées par des virgules (ou des points virgules si on ne veut pas voir le résultat des instructions d'affectations à l'écran). Cependant dans un script (ou une fonction), le passage à la ligne est équivalent à la virgule, ce qui permet une présentation de la forme :

  ```
  for  variable  =   matrice
     instruction1
     instruction2
     ...........
     instruction n
  end
  ```

  où les instructions peuvent être suivies d'un point virgule (toujours pour éviter les affichages à l'écran)[3].

3.1.2 La boucle `while`

Elle permet de répéter une suite d'instructions tant qu'une condition est vraie, par exemple :

```
-->x=1 ; while x<14,x=2*x, end
```

Signalons que les opérateurs de comparaisons sont les suivants :

==	égal à
<	strictement plus petit que
>	strictement plus grand que
<=	plus petit ou égal
>=	plus grand ou égal
~= ou <>	différent de

et que Scilab possède un type logique ou booléen : `%t` ou `%T` pour vrai et `%f` ou `%F` pour faux. On peut définir des matrices et vecteurs de booléens. Les opérateurs logiques sont :

&	et
|	ou
~	non

La syntaxe du `while` est la suivante :

```
while condition, instruction_1, ... ,instruction_N , end
```

ou encore (dans un script ou une fonction) :

```
while condition
   instruction_1
   .............
   instruction_N
end
```

où chaque `instruction_k` peut être suivie d'un point virgule et ce qui est appelé `condition` est en fait une expression délivrant un scalaire booléen.

3. ceci est uniquement valable pour un script car dans une fonction, le résultat d'une instruction d'affectation n'est pas affiché même si elle n'est pas suivie d'un point virgule ; ce comportement par défaut pouvant être modifié avec l'instruction `mode`

3.2 Les instructions conditionnelles

Il y en a aussi deux : un « if then else » et un « select case ».

3.2.1 La construction if then else

Voici un exemple :

```
-->if x>0 then, y=-x,else,y=x,end    // la variable x doit être définie
```

De même dans un script ou une fonction, si vous allez à la ligne, les virgules de séparation ne sont pas obligatoires. Comme pour les langages habituels, si aucune action n'intervient dans le cas où la condition est fausse, la partie `else, instructions` est omise. Enfin, si la partie `else` enchaîne sur un autre if then else, on peut lier les mots clés `else` et `if` ce qui conduit finalement à une présentation du type :

```
if condition_1  then
   suite   d'instructions  1
elseif condition_2  then
   suite d'instructions 2
.........................
elseif  condition_N  then
   suite d'instructions N
else
   suite d'instructions N+1
end
```

où, de même que pour le `while`, chaque `condition` est une expression délivrant un scalaire booléen.

3.2.2 La construction select case (*)

Voici un exemple (à tester avec différentes valeurs de la variable `num`)[4]

```
-->num = 1, select num, case 1, y = 'cas 1', case 2, y = 'cas  2',...
-->else, y = 'autre cas', end
```

qui dans un script ou une fonction s'écrirait plutôt :

```
// ici  on suppose que  la variable num  est bien definie
select  num
case 1 y = 'cas 1'
case 2 y = 'cas 2'
else   y = 'autre cas'
end
```

Ici, Scilab teste successivement l'égalité de la variable `num` avec les différents cas possibles (1 puis 2), et dès que l'égalité est vraie, les instructions correspondantes (au cas) sont effectuées puis on sort de la construction. Le `else`, qui est facultatif, permet de réaliser une suite d'instructions dans le cas où tous les précédents tests ont échoués. La syntaxe de cette construction est la suivante :

```
select variable_test
case expr_1
    suite d'instructions 1
...........................
case expr_N
    suite d'instructions N
else
    suite d'instructions N+1
end
```

où ce qui est appelé `expr_i` est une expression qui délivrera une valeur à comparer avec la valeur de la variable `variable_test` (dans la plupart des cas ces expressions seront des constantes ou des variables). En fait cette construction est équivalente à la construction `if` suivante :

```
if variable_test =  expr_1 then
```

4. la variable `y` est du type « chaîne de caractères », cf prochain paragraphe

```
   suite d'instructions  1
........................
elseif variable_test = expr_N then
   suite  d'instructions    N
else
   suite d'instructions N+1
end
```

3.3 Autres types de données

Jusqu'à présent nous avons vu les types de données suivants :

1. les matrices (et vecteurs et scalaires) de nombres réels[5] ou complexes ;
2. les booléens (matrices, vecteurs et scalaires) ;

Il en existe d'autres dont les chaînes de caractères et les listes.

3.3.1 Les chaînes de caractères

Dans l'exemple sur la construction `case`, la variable `y` est du type chaîne de caractères. Dans le langage Scilab elles sont délimitées par des apostrophes ou des guillemets (anglais), et lorsqu'une chaîne contient un tel caractère, il faut le doubler. Ainsi pour affecter à la variable `est_ce_si_sur`, la chaîne :

```
Scilab c'est "cool" ?
```

on utilisera :

```
-->est_ce_si_sur = "Scilab c''est ""cool"" ?"
```

ou bien :

```
-->est_ce_si_sur = 'Scilab c''est ""cool"" ?'
```

On peut aussi définir des matrices de chaînes de caractères :

```
-->Ms = ["a" "bc" "def"]
 Ms  =

!a  bc  def  !

-->size(Ms)  // pour obtenir les dimensions
 ans  =

!   1.    3. !

-->length(Ms)
 ans  =

!   1.    2.    3. !
```

Noter que `length` n'a pas le même comportement que sur une matrice de nombres : pour une matrice de chaînes de caractères `M`, `length(M)` renvoie une matrice d'entiers de même format que `M` où le coefficient en position (i,j) donne le nombre de caractères de la chaîne en position (i,j).

La concaténation de chaînes de caractères utilise simplement l'opérateur + :

```
-->s1 = 'abc'; s2 = 'def'; s = s1 + s2
s =

 abcdef
```

et l'extraction se fait via la fonction `part`:

```
-->part(s,3)
ans =
```

5. les entiers étant vu comme des nombres flottants


```
 c

-->part(s,3:4)
ans =

 cd
```

Le deuxième argument de la fonction `part` est donc un vecteur d'indices (ou un simple scalaire entier) désignant les numéros des caractères que l'on veut extraire.

3.3.2 Les listes (*)

Une liste est simplement une collection d'objets Scilab (matrices ou scalaires réels ou complexes, matrices ou scalaires « chaînes de caractères » , booléens, listes, fonctions, ...) numérotés. Il y a deux sortes de listes, les « ordinaires » et les « typées ». Voici un exemple de liste ordinaire :

```
-->L=list(rand(2,2),["Vivement que je finisse" " cette doc..."],[%t ; %f])
 L  =


       L(1)

!   0.2113249    0.0002211 !
!   0.7560439    0.3303271 !

       L(2)

!Vivement que je finisse   cette doc...  !

       L(3)

! T !
! F !
```

Je viens de définir une liste dont le premier élément est une matrice (2,2), le 2ème un vecteur de chaînes de caractères, et le 3ème un vecteur de booléens. Voici quelques opérations basiques sur les listes :

```
-->M = L(1)  // extraction de la premiere entree
 M  =

!   0.2113249    0.0002211 !
!   0.7560439    0.3303271 !

-->L(1)(2,2) = 100;  // modification dans la premiere entree

-->L(1)
 ans  =

!   0.2113249    0.0002211 !
!   0.7560439    100.      !

-->L(2)(4) = " avant les vacances !";  // je modifie la 2 eme entree

-->L(2)
 ans  =
```

```
!Vivement que je finisse   cette doc...      avant les vacances !  !

-->L(4)=" pour ajouter une 4 eme entree"
 L  =


       L(1)

!   0.2113249    0.0002211 !
!   0.7560439    100.      !

       L(2)

!Vivement que je finisse   cette doc...      avant les vacances !  !

       L(3)

! T !
! F !

       L(4)

  pour ajouter une 4 eme entree

-->size(L)   // quel est le nombre d'elements de la liste
 ans  =

    4.

-->length(L)  // idem
 ans  =

    4.

-->L(2) = null()  // destruction de la 2 eme entree
 L  =


       L(1)

!   0.2113249    0.0002211 !
!   0.7560439    100.      !

       L(2)

! T !
! F !

       L(3)

  pour ajouter une 4 eme entree

 -->Lbis=list(1,1:3)    // je definis une autre liste
 Lbis  =
```



```
       Lbis(1)

    1.

       Lbis(2)

!   1.    2.    3. !


-->L(3) = Lbis  // la 3 eme entree de L est maintenant une liste
 L  =


       L(1)

!   0.2113249    0.0002211 !
!   0.7560439    100.      !

       L(2)

! T !
! F !

       L(3)


        L(3)(1)

    1.

        L(3)(2)

!   1.    2.    3. !
```

Passons aux listes « typées » . Pour ces listes, le premier élément est une chaîne de caractères qui permet de « typer » la liste (ceci permet de définir un nouveau type de donnée puis de définir des opérateurs sur ce type), les éléments suivants pouvant être n'importe quels objets scilab. En fait, ce premier élément peut aussi être un vecteur de chaînes de caractères, la première donnant donc le type de la liste et les autres pouvant servir à référencer les différents éléments de la liste (au lieu de leur numéro dans la liste). Voici un exemple : on veut représenter un polyèdre (dont toutes les faces ont le même nombre d'arêtes). Pour cela, on stocke les coordonnées de tous les sommets dans une matrice (de format (3,nb sommets)) qui sera référencée par la chaîne *coord*. Puis on décrit par une matrice (de format (nb de faces, nb de sommets par face)) la connectivité de chacune des faces : pour chaque face, je donne les numéros des sommets qui la constituent de sorte à orienter la normale vers l'extérieur par la règle du tire-bouchon.

Cette entrée de la liste sera référencée par la chaîne *face*. Pour un cube (cf figure 3.1) cela peut donner :

```
-->P=[ 0 0 1 1 0 0 1 1;...     // les coordonnees des sommets
-->    0 1 1 0 0 1 1 0;...
-->    0 0 0 0 1 1 1 1];

-->connect=[ 1 2 3 4; 5 8 7 6; 3 7 8 4;...   // les faces
```

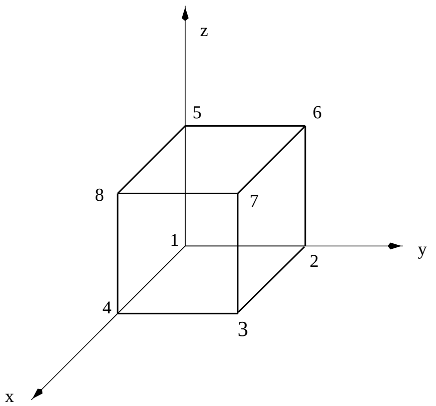


FIG. 3.1 – *Numéros des sommets du cube*

```
-->          2 6 7 3; 1 5 6 2; 1 4 8 5];
```

Il me reste à former ma liste typée qui contiendra donc toute l'information sur mon cube :

```
-->Cube = tlist(["polyedre","coord","face"],P,connect)
 Cube  =


       Cube(1)

!polyedre  coord  face  !

       Cube(2)

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

       Cube(3)

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

Au lieu de désigner les éléments constitutifs par leur numéro, on peut utiliser la chaîne de caractères correspondante, exemple :

```
-->Cube("coord")(:,2)
 ans  =

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



```
-->Cube("face")(1,:)
 ans  =

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

En dehors de cette particularité, ces listes typées se manipulent exactement comme les autres. Leur avantage est que l'on peut définir (i.e. surcharger) les opérateurs `+`, `-`, `/`, `*`, etc, sur une tlist de type donné (cf une prochaine version de cette doc ou mieux : consulter la doc en ligne sur la home page Scilab).

3.3.3 Quelques expressions avec les vecteurs et matrices de booléens (*)

Le type booléen se prête aussi à certaines manipulations matricielles dont certaines bénéficient de raccourcis d'écriture assez pratiques. Lorsque l'on compare deux matrices réelles de même taille avec l'un des opérateurs de comparaison (`<`, `>`, `<=`, `>=`,`==`, `~=`), on obtient une matrice de booléens de même format, la comparaison ayant lieu « élément par élément », par exemple :

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

!   0.2113249    0.0002211    0.6653811 !
!   0.7560439    0.3303271    0.6283918 !

-->B = rand(2,3)
 B  =

!   0.8497452    0.8782165    0.5608486 !
!   0.6857310    0.0683740    0.6623569 !

-->A < B
 ans  =

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

mais si l'une des deux matrices est un scalaire, alors `A <s` est un raccourci pour `A <s*one(A)` :

```
-->A < 0.5
 ans  =

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

Les opérateurs booléens s'appliquent aussi sur ce type de matrice toujours au sens « élément par élément » :

```
-->b1 = [%t %f %t]
 b1  =

! T F T !


-->b2 = [%f %f %t]
 b2  =

! F F T !

-->b1 & b2
 ans  =
```

```
! F F T !

-->b1 | b2
 ans  =

! T F T !

-->~b1
 ans  =

! F T F !
```

D'autre part il y a deux fonctions très pratiques qui permettent de vectoriser des tests :

1. `bool2s` transforme une matrice de booléens en une matrice de même format où la valeur logique « vraie » est transformée en 1, et « faux » en zéro :
   ```
   -->bool2s(b1)
    ans  =

   !   1.    0.    1. !
   ```
2. La fonction `find` permet de récupérer les indices des coefficients « vrais » d'un vecteur ou d'une matrice de booléens :
   ```
   -->v = rand(1,5)
    v  =

   !   0.5664249    0.4826472    0.3321719    0.5935095    0.5015342 !

   -->find(v < 0.5)  // v < 0.5 donne un vecteur de booleens
    ans  =

   !   2.    3. !
   ```

Une application de ces fonctions est exposée plus loin (cf Quelques remarques sur la rapidité). Enfin les fonctions `and` et `or` procèdent respectivement au produit logique (et) et à l'addition logique (ou) de tous les éléments d'une matrice booléenne. On obtient alors un scalaire booléen. Ces deux fonctions admettent un argument optionnel pour effectuer l'opération selon les lignes ou les colonnes.

3.4 Les fonctions

Pour définir une fonction en Scilab, la méthode la plus courante est de l'écrire dans un fichier, dans lequel on pourra d'ailleurs mettre plusieurs fonctions (en regroupant par exemple les fonctions qui correspondent à un même thème ou une même application). Chaque fonction doit commencer par l'instruction :

```
function [y1,y2,y3,...,yn]=nomfonction(x1,....,xm)
```

où les `xi` sont les arguments d'entrée, les `yj` étant les arguments de sortie. Il n'y a pas de mot clé délimitant la fin d'une fonction (genre `end` ou `end function`). Dans le cas où le fichier contient plusieurs fonctions, ce rôle est joué par l'instruction `function ...` de la fonction suivante, ou par la fin du fichier pour la dernière fonction). *Rmq :*

- la première ligne de votre fichier doit absolument commencer par l'instruction `function ...`, c-à-d que vous ne devez pas commenter la première fonction du fichier avant son entête ;
- la dernière instruction du fichier doit obligatoirement être suivie d'un passage à la ligne sinon l'interprèteur ne la prend pas en compte ;
- la tradition est de suffixer les noms des fichiers contenant des fonctions en `.sci`.

Voici un premier exemple :

```
function [y] = fact1(n)
  // la factorielle : il faut ici que n soit bien un entier naturel
  y = prod(1:n)
```

Supposons que l'on ait écrit cette fonction dans le fichier `facts.sci`. Pour que Scilab puisse la connaître, il faut charger le fichier par l'instruction :

```
getf('facts.sci')
```

ce qui peut aussi se faire via le menu `File operations` comme pour les scripts (après selection du fichier, il faut alors cliquer sur le bouton `getf`). On peut alors utiliser cette fonction à partir de l'invite (mais aussi dans un script ou bien une autre fonction) :

```
-->m = fact1(5)
 m  =

    120.

-->n1=2; n2 =3; fact1(n2)
 ans  =

    6.

-->fact1(n1*n2)
 ans  =

    720.
```

Avant de vous montrer un deuxième exemple, voici quelques précisions de vocabulaire. Dans l'écriture de la fonction, l'argument de sortie `y` et l'argument d'entrée `n` sont appelés *arguments formels* . Lorsque j'utilise cette fonction à partir de l'invite, d'un script ou d'une autre fonction :

```
arg_s = fact1(arg_e)
```

les arguments utilisés sont appelés *arguments effectifs*. Dans ma première utilisation, l'argument effectif d'entrée est une constante (5), dans le deuxième une variable (`n2`) et dans le troisième une expression (`n1*n2`). La correspondance entre arguments effectifs et formels (ce que l'on appelle couramment passage des paramètres) peut se faire de diverses manières (cf prochain paragraphe pour quelques précisions en ce qui concerne Scilab).

Voici un deuxième exemple : il s'agit d'évaluer en un point t un polynôme écrit dans une base de Newton (*Rmq :* avec les $x_i = 0$ on retrouve la base canonique) :

$$p(t) = c_1 + c_2(t - x_1) + c_3(t - x_1)(t - x_2) + \ldots + c_n(t - x_1)\ldots(t - x_{n-1}).$$

En utilisant les facteurs communs et en calculant de la « droite vers la gauche » (ici avec $n = 4$) :

$$p(t) = c_1 + (t - x_1)(c_2 + (t - x_2)(c_3 + (t - x_3)(c_4))),$$

on obtient l'algorithme d'Horner:

(1) $p := c_4$
(2) $p := c_3 + (t - x_3)p$
(3) $p := c_2 + (t - x_2)p$
(4) $p := c_1 + (t - x_1)p.$

En généralisant à n quelconque et en utilisant une boucle, on obtient donc en Scilab :

```
function [p]=myhorner(t,x,c)
```

```
  // evaluation du  polynome c(1) + c(2)*(t-x(1)) + c(3)*(t-x(1))*(t-x(2)) +
  //                          ... + c(n)*(t-x(1))*...*(t-x(n-1))
  // par l'algorithme d'horner
  n=length(c)
  p=c(n)
  for k=n-1:-1:1
     p=c(k)+(t-x(k))*p
  end
```

Si les vecteurs `coef` et `xx` et le réel `tt` sont bien définis dans l'environnement d'appel de la fonction (si le vecteur `coef` a m composantes, il faut que `xx` en ait au moins $m-1$, si l'on veut que tout se passe bien...), l'instruction :

```
val = myhorner(tt,xx,coef)
```

affectera à la variable `val` la valeur :

$$coef_1 + coef_2(tt - xx_1) + \cdots + coef_m \prod_{i=1}^{m-1}(tt - xx_i)$$

aux erreurs d'arrondi numérique près. Petit rappel : l'instruction `length` renvoie le produit des deux dimensions d'une matrice (de nombres), et donc dans le cas d'un vecteur (ligne ou colonne) son nombre de composantes. Cette instruction permet (avec l'instruction `size` qui renvoie le nombre de lignes et le nombre de colonnes) de ne pas faire passer la dimension des structures de données (matrices, listes, ...) dans les arguments d'une fonction.

3.4.1 Passage des paramètres (*)

Jusqu'à la version 2.3.1, toutes les variables d'entrée sont passées par valeur, c'est à dire qu'il y a une copie des arguments effectifs : si l'argument effectif est une matrice, l'argument formel correspondant est une copie de cette matrice. Vous pouvez donc modifier les variables d'entrée sans aucune conséquence sur les variables d'appel correspondantes. Ainsi dans l'exemple donné, on pourrait rajouter à la fin l'instruction :

```
c=ones(c)
```

qui modifierait localement la valeur du tableau `c` mais n'aurait aucune incidence sur le tableau `coef` de l'environnement d'appel. À partir de la version 2.4, les variables d'entrée sont « passées » par référence[6] sauf si elles sont modifiées dans la fonction. Dans ce cas, je pense que les variables d'entrée susceptibles d'être modifiées sont toujours passées par valeur (ceci est une supposition de ma part d'après ce que j'ai pu observer), et l'on a le même comportement qu'avec la version précédente (pas de modif sur les variables d'appel). Autre point : dans une fonction, vous avez accès (en lecture uniquement) à toutes les variables des niveaux supérieurs. On peut ainsi se permettre l'utilisation (en lecture) de variables globales quoique cela ne soit pas trop recommandable (jusqu'à Scilab 2.3.1 cette pratique permettait de ne pas trop consommer de mémoire pour les grosses structures de données d'une application car le passage par valeur à le défaut de consommer de la mémoire (puisqu'il y a duplication des données), et du temps (pour les copies). Si par contre vous modifiez une variable globale, une nouvelle variable interne (à la fonction) est créée, la variable de même nom du niveau supérieur n'étant toujours pas modifiée. Voici un exemple : on considère la fonction suivante :

```
function  [y1] = test(x1)
  y1 =  x1 + c    // on  utilise la variable  globale c (si elle existe ...)
  disp(c,'c  = ') // disp permet  de visualiser des variables  à l'ecran
  c(5,2) = 2      // une variable interne c est creee
  disp(c,'c = ')
```

6. pour une matrice, on peut fournir simplement l'adresse du premier élément ainsi que le nombre de lignes et de colonnes (et une indication précisant qu'il s'agit bien d'une matrice)

utilisée de la façon suivante :

```
-->c=0:0.1:1; x= ones(c);
-->y = test(x);

 c =

!  0.   0.1   0.2   0.3   0.4   0.5   0.6   0.7   0.8   0.9   1. !

 c =

!  0.   0. !
!  0.   0. !
!  0.   0. !
!  0.   0. !
!  0.   2. !

-->c       // elle n'a pas du etre modifiee ?

 c =

!  0.   0.1   0.2   0.3   0.4   0.5   0.6   0.7   0.8   0.9   1. !

-->clear c // effacement de la variable globale c

-->y = test(x);

 !--error 4 undefined  variable : c at  line 2 of function test  called by : y = test(x);
```

Enfin, lors de la sortie de la fonction, toutes les variables internes (donc propres à la fonction) sont détruites.

En conclusion :

1. ne pas utiliser de variables globales[7] (ça ne sert plus à rien depuis la version 2.4) ;
2. ne pas modifier les variables d'entrée (sauf celles qui font aussi parties des arguments de sortie...) ;
3. du point de vue de l'utilisateur, le saut de la version 2.3.1 à la 2.4 est indolore en ce qui concerne ce changement majeur dans le passage des paramètres et on consomme moins de mémoire intermédiaire ! Un exemple : soit la fonction :

   ```
   function [y] = test_bis(x)
     // une fonction qui ne sert a rien.
     y = 2*x
   ```

   Avec Scilab2.3 :

   ```
   -->x = rand(500,500);

   -->y = test_bis(x);

    !--error    17
   stack size exceeded! (Use stacksize function to increase it)
   Memory used for variables :   754136
   Intermediate memory needed:   250002
   Total  memory available   :  1000001
   at line       5 of function test_bis                 called by :
   y = test_bis(x);
   ```

7. Dans certains cas il est plus simple d'utiliser des variables globales et la dernière version de scilab (2.5) apporte des améliorations importantes pour les gérer : pour les détails cf `global` dans le Help.

alors que ça passe pour Scilab2.4 sans augmenter la taille de la pile (*Rappel :* pour augmenter cette dernière utiliser `stacksize(nb_mots)`).

3.4.2 Déverminage d'une fonction

Pour déboguer une fonction on peut déjà en premier lieu utiliser la fonction `disp(v1,v2, ...)` qui permet de visualiser la valeur des variables `v1`, `v2`,... dans l'ordre inverse (c'est pour cette raison que la chaîne de caractères `'c = '` dans l'instruction `disp(c,'c = ')` de l'exemple précédent a été mise en 2$^{\text{ème}}$ position). Dans un deuxième temps, vous pouvez mettre une ou plusieurs instruction(s) `pause` en des endroits stratégiques de la fonction. Lorsque Scilab rencontre cette instruction le déroulement du programme s'arrête et vous pouvez examiner la valeur de toutes les variables déjà définies à partir de la fenêtre Scilab (l'invite `-->` de Scilab se transforme en `-1->`). Lorsque vos observations sont finies, la commande `resume` fait repartir le déroulement des instructions (jusqu'à l'éventuelle prochaine `pause`).

3.4.3 L'instruction `break`

Elle permet dans une boucle `for` ou `while` d'arrêter le déroulement des itérations en passant le contrôle à l'instruction qui suit le `end` marquant la fin de la boucle[8]. Elle peut servir à simuler les autres types de boucles, celles avec le test de sortie à la fin (genre `repeat ... until` du Pascal) et celles avec test de sortie au milieu (arg...) ou bien à traiter les cas exceptionnels qui interdisent le déroulement normal d'une boucle `for` ou `while` (par exemple un pivot quasi nul dans une méthode de Gauss). Supposons que l'on veuille simuler une boucle avec test de sortie à la fin :

```
répéter
  suite d'instructions
jusqu'à ce que condition
```

où `condition` est une expression qui délivre un scalaire booléen (on sort lorsque ce test est vrai). On pourra alors écrire en Scilab :

```
while %t    // début de la boucle
  suite d'instructions
  if condition then, break, end
end
```

Il y a aussi des cas où l'utilisation d'un `break` conduit à une solution plus naturelle, lisible et compacte. Voici un exemple : on veut rechercher dans un vecteur de chaînes de caractères l'indice du premier mot qui commence par une lettre donnée l. Pour cela, on va écrire une fonction (qui renvoie 0 dans le cas où aucune des chaînes de caractères ne commenceraient par la lettre en question). En utilisant une boucle `while` (sans s'autoriser de `break`), on peut être conduit à la solution suivante :

```
function ind = recherche2(v,l)
  n = max(size(v))
  i = 1
  succes = %f
  while ~succes & (i <=  n)
     succes = part(v(i),1)  == l
     i =  i + 1
  end
  if  succes then
     ind = i-1
  else
     ind = 0
  end
```

Si on s'autorise l'utilisation d'un `break`, on a la solution suivante plus naturelle (mais moins conforme aux critères purs et durs de la programmation structurée) :

```
function ind  = recherche1(v,l)
```

8. si la boucle est imbriquée dans une autre, `break` permet de sortir uniquement de la boucle interne


```
  n   = max(size(v))
  ind =  0
  for i=1:n
     if part(v(i),1) == l then
        ind = i
        break
     end
  end
```

Rappel : on peut remarquer l'emploi de la fonction `size` alors que la fonction `length` semble plus adaptée pour un vecteur[9] ; ceci vient du fait que `length` réagit différemment si les composantes de la matrice ou du vecteur sont des chaînes de caractères (`length` renvoie une matrice de taille identique où chaque coefficient donne le nombre de caractères de la chaîne correspondante).

3.4.4 Quelques primitives utiles dans les fonctions

En dehors de `length` et `size` qui permettent de récupérer les dimensions des structures de données, et de `pause`, `resume`, `disp` qui permettent de déboguer, d'autres fonctions peuvent être utiles comme `error`, `warning`, `argn` ou encore `type` et `typeof`.

La fonction `error`

Elle permet d'arrêter brutalement le déroulement d'une fonction tout en affichant un message d'erreur ; voici une fonction qui calcule $n!$ en prenant soin de vérifier que $n \in \mathbb{N}$:

```
function [f] = fact2(n)
  // calcul de la factorielle d'un nombre entier positif
  if (n - floor(n) ~=0) | n<0 then
     error('erreur dans fact2  : l''argument doit etre un nombre entier naturel')
  end
  f = prod(1:n)
```

et voici le résultat sur deux arguments :

```
-->fact2(3) ans =

   6.

-->fact2(0.56)

 !--error 10000 erreur dans fact2 : l'argument doit etre un nombre entier naturel
 at line 6 of function fact called by : fact(0.56)
```

La fonction `warning`

Elle permet d'afficher un message à l'écran mais n'interrompt pas le déroulement de la fonction :

```
function [f] = fact3(n)
  // calcul de la factorielle d'un nombre entier positif
  if (n - floor(n) ~=0) | n<0  then
     n = floor(abs(n))
     warning('l''argument n''est pas un entier naturel: on calcule '+sprintf("%d",n)+"!")
  end
  f = prod(1:n)
```

9. si l'on veut un code qui fonctionne indépendamment du fait que `v` soit ligne ou colonne, on ne peut pas non plus utiliser `size(v,'r')` ou `size(v,'l')` d'où le `max(size(v))`

ce qui donnera par exemple :

```
-->fact3(-4.6)
WARNING:l'argument n'est pas un entier naturel: on calcule 4!
 ans  =

    24.
```

Comme pour la fonction `error`, l'argument unique de la fonction `warning` est une chaîne de caractères. J'ai utilisé ici une concaténation de 3 chaînes dont l'une est obtenue par la fonction `sprintf` qui permet de transformer des nombres en chaîne de caractères selon un certain format.

Les fonctions `type` et `typeof`

Celles-ci permettent de connaître le type d'une variable `v`. `type(v)` renvoie un entier alors que `typeof(v)` renvoie une chaîne de caractères Voici un tableau récapitulatif pour les types de données que nous avons vus :

type de `v`	`type(v)`	`typeof(v)`
matrice de réels ou complexes	1	constant
matrice de booléens	4	boolean
matrice de chaînes de caractères	10	string
liste	15	list
liste typée	16	type de la liste
fonction	13	function

Pour les listes typées, `typeof` ne fonctionne pas sans une petite modification qui est simple à effectuer si vous avez le code source de Scilab (cf archivage des messages du forum sur la home page Scilab). Un exemple d'utilisation : on veut sécuriser notre fonction factorielle en cas d'appel avec un mauvais argument d'entrée.

```
function [f] = fact4(n)
  // la fonction factorielle un peu plus blindee
  if type(n) ~= 1 then
    error(" erreur dans fact4 : l''argument n''a pas le bon type...")
  end
  [nl,nc]=size(n)
  if (nl ~= 1) | (nc ~= 1) then
    error(" erreur dans fact4 : l''argument ne doit pas etre une matrice...")
  end
  if (n - floor(n) ~=0) | n<0  then
     n = floor(abs(n))
     warning('l''argument n''est pas un entier naturel: on calcule '+sprintf("%d",n)+"!")
  end
  f = prod(1:n)
```

La fonction `argn`

Elle permet d'obtenir le nombre d'arguments effectifs d'entrée et de sortie d'une fonction lors d'un appel à celle-ci. On l'utilise sous la forme :

```
[lhs,rhs] = argn(0)
```

`lhs` (pour left hand side) donnant le nombre d'arguments de sortie effectifs, et `rhs` (pour right hand side) donnant le nombre d'arguments d'entrée effectifs.

Elle permet essentiellement d'écrire une fonction avec des arguments d'entrée et de sortie optionnels (la fonction `type` ou `typeof` pouvant d'ailleurs l'aider dans cette tache). Un exemple d'utilisation est donné plus loin (Une fonction est une variable Scilab).

3.5 Compléments divers

3.5.1 Longueur des identificateurs

Scilab ne prend en compte que des 24 premières lettres de chaque identificateur :

```
-->a2345678901234567890123456789012345 = 1
 a234567890123456789012345678901234  =

    1.
-->a2345678901234567890123456789012345
 a234567890123456789012345678901234  =

    1.
```

On peut donc utiliser plus de lettres mais seules les 24 premières sont significatives.

3.5.2 Priorité des opérateurs

Elle est assez naturelle (c'est peut être la raison pour laquelle ces règles n'apparaissent pas dans l'aide en ligne...). Comme usuellement les opérateurs numériques[10] (`+ - * .* / ./ \ ^ .^ '`) ont une priorité plus élevée que les opérateurs de comparaisons (`< <= > >= == ~=`). Voici un tableau récapitulatif par ordre de priorité décroissante pour les opérateurs numériques :

`'`
`^ .^`
`* .* / ./ \`
`+ -`

Pour les opérateurs booléens le « non » (`~`) est prioritaire sur le « et » (`&`) lui-même prioritaire sur le « ou » (`|`). Lorsque l'on ne souvient plus des priorités, on met des parenthèses ce qui peut d'ailleurs aider (avec l'ajout de caractères blancs) la lecture d'une expression comportant quelques termes... Pour plus de détails voir le « Scilab Bag Of Tricks ». Quelques remarques :

1. Comme dans la plupart des langages, le `-` unaire n'est autorisé qu'en début d'expression, c-à-d que les expressions du type (où *op* désigne un opérateur numérique quelconque) :

   opérande op - opérande

   sont interdites. Il faut alors mettre des parenthèses :

   opérande op (- opérande)

2. Pour une expression du type :

   opérande op1 opérande op2 opérande op3 opérande

   où les opérateurs ont une priorité identique, l'évaluation se fait en général de la gauche vers la droite :

   ((opérande op1 opérande) op2 opérande) op3 opérande

   sauf pour l'opérateur d'élévation à la puissance :

   `a^b^c` est évalué de la droite vers la gauche : `a^(b^c)`

   de façon à avoir la même convention qu'en mathématique : a^{b^c}.

3. Contrairement au langage C, l'évaluation des expressions booléennes de la forme :

   a ou b
   a et b

10. il y en a d'autres que ceux qui figurent dans cette liste

passe d'abord par l'évaluation des sous-expressions booléennes a et b avant de procéder au « ou » pour la première ou au « et » pour la deuxième (dans le cas où a renvoie « vrai » pour la première (et faux pour la deuxième) on peut se passer d'évaluer l'expression boolénne b). Ceci interdit certains raccourcis utilisés en C. Par exemple le test dans la boucle suivante :

```
while i>0 & temp < v(i)
  v(i+1) = v(i)
  i = i-1
end
```

(où `v` est un vecteur et `temp` un scalaire), générera une erreur pour $i = 0$ car la deuxième expression `temp < v(i)` sera quand même évaluée (les vecteurs étant toujours indicés à partir de 1 !) :

3.5.3 Récursivité

Une fonction peut s'appeler elle-même. Voici deux exemples très basiques (le deuxième illustrant une mauvaise utilisation de la récursivité)

```
function f=fact(n)
   // la factorielle en recursif
   if n <= 1 then
      f = 1
   else
      f = n*fact(n-1)
   end

function f=fib(n)
   // calcul du n ieme terme de la suite de Fibonnaci :
   // fib(0) = 1, fib(1) = 1, fib(n+2) = fib(n+1) + fib(n)
   if n <= 1 then
      f = 1
   else
      f = fib(n-1) + fib(n-2)
   end
```

3.5.4 Une fonction est une variable Scilab

Une fonction programmée en langage Scilab [11] est une variable du type « function » , et en particulier, elle peut être passée comme argument d'une autre fonction. Voici un petit exemple [12] : ouvrez un fichier pour écrire les deux fonctions suivantes :

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

function dessine_fonction(a, b, fonction, n)
  // n est un argument optionnel, en cas d'absence de celui-ci on impose n=61
  [lhs, rhs] = argn(0)
  if rhs == 3 then
     n = 61
  end
  x = linspace(a,b,n)
  y = fonction(x)
  plot(x,y)
```

puis rentrez ces fonctions dans l'environnement et enfin, essayez par exemple :

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

11. voir la section « Primitives et fonctions Scilab » du bétisier
12. qui est inutile en dehors de son aspect pédagogique : cf `fplot2d`

Une possibilité intéressante de Scilab est que l'on peut définir directement une fonction dans l'environnement (sans passer par l'écriture d'un fichier puis le chargement par `getf`) par l'intermédiaire de la commande `deff` dont voici la syntaxe simplifiée :

```
deff('[y1,y2,...]=nom_de_la_fonction(x1,x2,....)',text)
```

où `text` est un vecteur colonne de chaînes de caractères, constituant les instructions successives de la fonction. Par exemple, on aurait pu utiliser :

```
deff('[y]=f(x)','y = sin(x).*exp(-abs(x))')
```

pour définir la première des deux fonctions précédentes. En fait cette possibilité est intéressante dans plusieurs cas :

1. dans un script utilisant une fonction simple qui doit être modifiée assez souvent ; dans ce cas on peut définir la fonction avec `deff` dans le script ce qui évite de jongler avec un autre fichier dans lequel on aurait défini la fonction comme d'habitude : avec la méthode classique il faut penser à recharger le fichier contenant la fonction par `getf` à chaque modification ; bien que cette opération puisse s'automatiser en écrivant l'ordre `getf` dans le script, je pense que la méthode avec le `deff` est plus appropriée si le code associé à la fonction (ou les fonctions) est court : tout est dans un seul et même fichier (je pense ici à des petites applications) ;
2. la possibilité vraiment intéressante est de pouvoir définir une partie du code de façon dynamique : on élabore une fonction à partir d´éléments divers issus de calculs précédents et/ou de l'introduction de données (via un fichier ou de façon interactive (cf Fenêtres de dialogues)) ; dans cet esprit voir aussi les fonctions `evstr` et `execstr` un peu plus loin.

3.5.5 Fenêtres de dialogues

Dans l'exemple de script donné dans le chapitre 2, on a vu la fonction `input` qui permet de rentrer un paramètre interactivement via la fenêtre Scilab. D'autre part la fonction `disp` permet l'affichage à l'écran de variables (toujours dans la fenêtre Scilab). En fait il existe une série de fonctions qui permettent d'afficher des fenêtres de dialogues, menus, selection de fichiers : `x_choices` , `x_choose`, `x_dialog`, `x_matrix`, `x_mdialog`, `x_message` et `xgetfile`. Voir le `Help` pour le détail de ces fonctions (l'aide sur une fonction propose toujours au moins un exemple).

3.5.6 Conversion d'une chaîne de caractères en expression Scilab

Il est souvent utile de pouvoir évaluer une expression Scilab mise sous la forme d'une chaîne de caractères. Par exemple, la plupart des fonctions précédentes renvoient des chaînes de caractères, ce qui s'avère pratique même pour rentrer des nombres car on peut alors utiliser des expressions Scilab (exemples `sqrt(3)/2, 2*%pi`, ...). L'instruction qui permet cette conversion est `evstr`, exemple :

```
-->c = "sqrt(3)/2"
 c  =
 sqrt(3)/2

-->d = evstr(c)
 d  =
    0.8660254
```

Dans la chaîne de caractères vous pouvez utiliser des variables Scilab déjà définies :

```
-->a = 1;
-->b=evstr("2 + a")
```

```
 b  =
    3.
```

et cette fonction s'applique aussi sur une matrice de chaînes de caractères [13] :

```
-->evstr(["a" "2" ])
 ans  =

!   1.    2. !

-->evstr([" a + [1 2]" "[4 , 5]"])
 ans  =

!   2.    3.    4.    5. !

-->evstr(["""a""" """b"""])   // conversion d'une chaine en une chaine
 ans  =

!a  b  !
```

Il existe aussi la fonction `execstr` qui permet d'exécuter une instruction Scilab donnée sous forme d'une chaîne de caractères :

```
-->execstr("A=rand(2,2)")

-->A
 A  =

!   0.2113249    0.0002211 !
!   0.7560439    0.3303271 !
```

3.5.7 Lecture/écriture sur fichiers

Dans le chapitre deux, nous avons vu comment lire et écrire une matrice de réels dans un fichier en une seule instruction avec `read` et `write`. De même il est possible d'écrire et de lire un vecteur colonne de chaînes de caractères :

```
-->v = ["Scilab is free";"Octave is free";"Matlab is ?"];

-->write("toto.dat",v,"(A)")  // aller voir le contenu du fichier toto.dat

-->w = read("toto.dat",-1,1,"(A)")
 w  =

!Scilab is free  !
!                !
!Octave is free  !
!                !
!Matlab is ?     !
```

Pour l'écriture, on rajoute simplement un troisième argument à `write` qui correspond à un format fortran : c'est une chaîne de caractères comprenant un (ou plusieurs) descripteur(s) d'édition (séparés par des virgules s'il y a en plusieurs) entourés par des parenthèses : le `A` signifie que l'on souhaite écrire une chaîne de caractères. Pour la lecture, les deuxième et troisième arguments correspondent respectivement au nombre de lignes (-1 pour aller jusqu'à la fin du fichier) et colonne (ici 1). En fait pour les matrices de réels vous pouvez aussi rajouter un format (plutôt en écriture) de façon à contrôler précisemment la façon dont seront écrites les données.

13. et aussi sur une liste, voir le Help

D'une manière générale les possibilités de Scilab en ce domaine sont exactement celle du fortran 77, vous pouvez donc lire un livre sur ce langage pour en savoir plus [14]. Dans la suite je vais simplement donner quelques exemples concernant uniquement les fichiers « texte » à accès séquentiel.

Ouvrir un fichier

Cela se fait avec l'instruction `file` dont la syntaxe (simplifiée) est :

```
[unit, [err]]=file('open', file-name ,[status])
```

où :

- `file-name` est une chaîne de caractère donnant le nom du fichier (éventuellement précédée du chemin menant au fichier si celui-ci ne se trouve pas dans le répertoire pointé par Scilab, ce répertoire se changeant avec l'instruction `chdir`) ;
- `status` est l'une des chaînes de caractères :
  - `"new"` pour ouvrir un nouveau fichier (si celui-ci existe déjà une erreur est générée) ;
  - `"old"` pour ouvrir un fichier existant (si celui-ci n'existe pas une erreur est générée) ;
  - `"unknow"` si le fichier n'existe pas, un nouveau fichier est créé, et dans le cas contraire le fichier correspondant est ouvert ;

  Dans le cas où `status` n'est pas présent, Scilab utilise `"new"` (c'est pour cette raison que l'écriture d'un fichier en une seule instruction `write` échoue si le fichier existe déjà).
- `unit` est un entier qui va permettre d'identifier le fichier par la suite dans les opérations de lectures/écritures (plusieurs fichiers pouvant être ouverts en même temps).
- Une erreur à l'ouverture d'un fichier peut être détectée si l'argument `err` est présent ; dans le cas contraire, Scilab gère l'erreur brutalement. Une absence d'erreur correspond à la valeur 0 et lorsque cette valeur est différente, l'instruction `error(err)` renvoie un message d'erreur permettant d'en savoir un peu plus : on obtient en général err=240 ce qui signifie :

  ```
      -->error(240)
                 !--error   240
      File error(240) already exists or directory write access denied
  ```

  Pour permettre de récupérer un nom de fichier de façon interactive on utilisera `xgetfile` qui permet de naviguer dans l'arborescence pour selectionner un fichier.

Écrire et lire dans le fichier ouvert

Supposons donc que l'on ait ouvert un fichier avec succès : celui-ci est repéré par l'entier `unit` qui nous a été renvoyé par `file`. Si le fichier existait déjà, les lectures/écritures ont normalement lieu en début de fichier. Si vous voulez écrire à la fin du fichier, il faut s'y positionner au préalable avec l'instruction `file("last", unit)`, et si pour une raison quelconque vous voulez revenir en début de fichier, on utilise `file("rewind", unit)`.

Voici un premier exemple : on veut écrire un fichier qui permet de décrire une liste d'arêtes du plan, c'est à dire que l'on considère n points $P_i = (x_i,y_i)$ et m arêtes, chaque arête étant décrite comme un segment $\overrightarrow{P_iP_j}$, et l'on donnera simplement le numéro (dans le tableau des points) du point de départ (ici i) puis celui d'arrivée (ici j). On choisit comme format pour ce fichier, la séquence suivante :

```
une ligne de texte
n
x_1 y_1
.......
```

14. Vous pouvez récupérer gratuitement le livre de Clive Page sur le serveur ftp `ftp.star.le.ac.uk` : se positionner dans le répertoire `/pub/fortran` et récupérer le fichier `prof77.ps.gz`

```
x_n y_n
m
i1 j1
.....
im jm
```

La ligne de texte permet de mettre quelques informations. On a ensuite un entier donnant le nombre de points, puis les coordonnées de ces points. Vient ensuite le nombre d'arêtes puis la connectivité de chaque arête. Supposons que notre ligne de texte soit contenue dans la variable `texte`, nos points dans la matrice `P` de format $(n,2)$ et la connectivité des arêtes dans la matrice `connect` de format $(m,2)$, l'écriture du fichier s'effectue avec les instructions :

```
write(unit,texte)      // ecriture de la ligne de texte
write(unit,size(P,1))  // ecriture du nombre de points
write(unit,P)          // ecriture des coordonnees des points
write(unit,size(connect,1)) // ecriture du nombre d'aretes
write(unit,connect)         // ecriture de la connectivite
file("close",unit)          // fermeture du fichier
```

et voici le résultat obtenu :

```
 un polygone au hasard
     5.0000000000000
    0.28553641680628   0.64885628735647
    0.86075146449730   0.99231909401715
    0.84941016510129    5.0041977781802D-02
    0.52570608118549   0.74855065811425
    0.99312098976225   0.41040589986369
     5.0000000000000
     1.0000000000000    2.0000000000000
     2.0000000000000    3.0000000000000
     3.0000000000000    4.0000000000000
     4.0000000000000    5.0000000000000
     5.0000000000000    1.0000000000000
```

qui n'est pas très harmonieux car nous n'avons pas précisé de formats d'édition. Le point négatif est que les entiers étant considérés comme des flottants par Scilab[15], ils sont écrits avec un format relatif aux flottants. D'autre part, la chaîne de caractères est précédée d'un blanc (non contenu dans la chaîne `texte`). Pour obtenir quelque chose de mieux, il faut rajouter ces formats fortran :

- pour un entier, on utilise `Ix` où `x` est un entier strictement positif donnant la longueur du champ en nombre de caractères (le cadrage a lieu à droite) ;
- pour les flottants, un format passe partout est `Ex.y` où `x` est la longueur totale du champ et `y` la longueur de la mantisse, la sortie prenant la forme : `[signe]0.mantisseE[signe]exposant` ; pour les flottants double précision, la conversion en décimal donne environ 16 chiffres significatifs et les exposants sont compris (environ) entre -300 et +300, ce qui donne une longueur totale de 24 caractères. On peut donc utiliser le format `E24.16` (selon la magnitude d'un nombre et la présentation désirée d'autres formats seraient plus adaptés) ;
- pour éviter le blanc précédant la chaîne de caractère, on peut utiliser le format `A`.

En reprenant l'exemple précédent, une écriture plus harmonieuse est obtenue avec (en supposant moins de 999 points et arêtes) :

```
write(unit,texte,"(A)")            // ecriture de la ligne de texte
write(unit,size(P,1),"(I3)")       // ecriture du nombre de points
```

15. Depuis la version 2.5, il existe cependant les types entiers int8, int16 et int32 voir le `Help`.


```
write(unit,P,"(2(X,E24.16))")      // ecriture des coordonnees des points
write(unit,size(connect,1),"(I3)") // ecriture du nombre d'aretes
write(unit,connect,"(2(X,I3))")    // ecriture de la connectivite
file("close",unit)                 // fermeture du fichier
```

(le format `X` correspond à l'écriture d'un caractère blanc et j'utilise aussi un facteur de répétition, `2(X,E24.16)` signifiant que l'on veut écrire sur une même ligne deux champs comprenant un blanc suivi d'un flottant écrit sur 24 caractères) ce qui donne :

```
un polygone au hasard
  5
   0.2855364168062806E+00   0.6488562873564661E+00
   0.8607514644972980E+00   0.9923190940171480E+00
   0.8494101651012897E+00   0.5004197778180242E-01
   0.5257060811854899E+00   0.7485506581142545E+00
   0.9931209897622466E+00   0.4104058998636901E+00
  5
   1   2
   2   3
   3   4
   4   5
   5   1
```

Pour lire ce même fichier, on pourrait utiliser la séquence suivante :

```
texte=read(unit,1,1,"(A)")      // lecture de la ligne de texte
n = read(unit,1,1)              // lecture du nombre de points
P = read(unit,n,2)              // lecture des coordonnees des points
m = read(unit,1,1)              // lecture du nombre d'aretes
connect = read(unit,m,2)        // lecture de la connectivite
file("close",unit)              // fermeture du fichier
```

Si vous avez bien lu ces quelques exemples, vous avez du remarquer que la fermeture d'un fichier s'obtient avec l'instruction `file("close",unit)`.

Pour finir, vous pouvez lire et écrire dans la fenêtre Scilab en utilisant respectivement `unit = %io(1)` et `unit = %io(2)`. Pour l'écriture, on peut alors obtenir une présentation plus soignée que celle obtenue avec la fonction `disp` (voir un exemple dans le Bétisier dans la section « Primitives et fonctions Scilab » (script d'appel à la fonction MonteCarlo)).

3.5.8 Remarques sur la rapidité

Voici sur deux exemples quelques trucs à connaître. On cherche à calculer une matrice de type Vandermonde :

$$A = \begin{bmatrix} 1 & t_1 & t_1^2 & \dots & t_1^n \\ 1 & t_2 & t_2^2 & \dots & t_2^n \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & t_m & t_m^2 & \dots & t_m^n \end{bmatrix}$$

Voici un premier code assez naturel :

```
function A=vandm1(t,n)
  // calcul de la matrice de Vandermonde A=[a(i,j)] 1<= i <= m
  //                                                1<= j <= n+1
  // ou a(i,j) = ti^(j-1)
  // t doit etre un vecteur colonne a m composantes
  m=size(t,'r')
  for i = 1:m
```

```
    for j = 1:n+1
      A(i,j) = t(i)^(j-1)
    end
  end
```

Comme a priori on ne déclare pas les tailles des matrices et autres objets en Scilab nul besoin de lui dire que le format final de notre matrice A est $(m,n+1)$. Comme au fur et à mesure du calcul la matrice grossie, Scilab doit gérer ce problème (pour $i = 1$, `A` est un vecteur ligne à j composantes, pour $i > 1$, `A` est une matrice $(i,n+1)$, on a donc en tout $n+m-1$ changements dans les dimensions de `A`. Par contre si on fait une pseudo-déclaration de la matrice (par la fonction `zeros(m,n+1)`) :

```
function A=vandm2(t,n)
  // idem a vandm1 sauf que l'on fait une pseudo-declaration
  // pour A
  m=size(t,'r')
  A = zeros(m,n+1)     // pseudo declaration
  for i = 1:m
    for j = 1:n+1
      A(i,j) = t(i)^(j-1)
    end
  end
```

il n'y a plus ce problème et l'on gagne un peu de temps (en fait MATLAB est plus sensible à ce genre d'optimisation) :

```
-->t = linspace(0,1,600)';
-->timer(); A = vandm1(t,100); timer()
 ans  =

    11.96

-->timer(); A = vandm2(t,100); timer()
 ans  =

    9.54
```

On peut essayer d'optimiser un peu ce code en initialisant A avec `ones(m,n+1)` (ce qui évite le calcul de la première colonne), en ne faisant que des multiplications avec $a_{ij} = a_{ij-1} \times t_i$ (ce qui évite les calculs de puissances), voire en inversant les deux boucles, mais l'on gagne peu. La bonne méthode est de voir que la construction de A peut se faire en utilisant une instruction vectorielle :

```
function A=vandm5(t,n)
  // la bonne methode : utiliser l'ecriture matricielle
  m=size(t,'r')
  A=ones(m,n+1)
  for i=1:n
     A(:,i+1)=t.^i
  end

function A=vandm6(t,n)
  // idem a vandm5 avec une petite optimisation
  m=size(t,'r')
  A=ones(m,n+1)
  for i=1:n
```



```
     A(:,i+1)=A(:,i).*t
  end
```

et on améliore ainsi la rapidité de façon significative :

```
-->timer(); A = vandm5(t,100); timer()
 ans  =

    0.11

-->timer(); A = vandm6(t,100); timer()
 ans  =

    0.06
```

Voici un deuxième exemple : il s´agit d´évaluer en plusieurs points (des scalaires réels mis dans un vecteur ou une matrice) la fonction « chapeau » (cf fig (3.2) :

$$\phi(t) = \begin{cases} 0 \text{ si } t \leq a \\ \frac{t-a}{b-a} \text{ si } a \leq t \leq b \\ \frac{c-t}{c-b} \text{ si } b \leq t \leq c \\ 0 \text{ si } t \geq c \end{cases}$$

Du fait de la définition par morceaux de cette fonction, son évaluation en un point nécessite en général

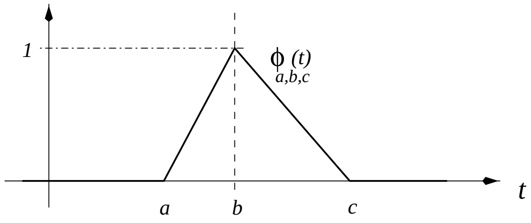


FIG. 3.2 – *Fonction « chapeau »*

plusieurs tests. Si ce travail doit être réalisé sur beaucoup de points il faut « vectoriser » ces tests pour éviter de faire travailler l´interpréteur. Par exemple, ce premier code naturel :

```
function [y]=phi1(t,a,b,c)
  // evalue la fonction chapeau (de parametres a, b et c) sur le vecteur
  // (voire la matrice) t au sens << element par element >>.
  // a,b et c doivent verifier a < b < c
  [n,m] = size(t)
  y = zeros(t)
  for j=1:m, for i=1:n
     if t(i,j) > a then
        if t(i,j) < b then
           y(i,j) = (t(i,j) - a)/(b - a)
        elseif t(i,j) < c then
           y(i,j) = (c - t(i,j))/(c - b)
        end
     end
  end, end
```

donne le résultat :

```
-->a = -0.2 ; b=0 ; c=0.15;

-->t = rand(100000,1)-0.5;

-->timer(); y1 = phi1(t,a,b,c); timer()
 ans  =

    66.7
```

alors que les codes suivants[16] :

```
function [y]=phi2(t,a,b,c)
  // evalue la fonction chapeau (de parametres a, b et c) sur le scalaire t
  // ou le vecteur (voire la matrice) t au sens << element par element >>.
  // a,b et c doivent verifier a < b < c
  Indicatrice_a_b = bool2s( (a < t) & (t <= b) )
  Indicatrice_b_c = bool2s( (b < t) & (t < c ) )
  y = Indicatrice_a_b .* (t - a)/(b - a) + Indicatrice_b_c .* (c - t)/(c - b)

function [y]=phi3(t,a,b,c)
  // idem a phi2 avec une petite optimisation
  t_le_b = ( t <= b )
  Indicatrice_a_b = bool2s( (a < t) & t_le_b )
  Indicatrice_b_c = bool2s( ~t_le_b & (t < c) )
  y = Indicatrice_a_b .* (t - a)/(b - a) + Indicatrice_b_c .* (c - t)/(c - b)
```

sont plus rapides :

```
-->timer(); y2 = phi2(t,a,b,c); timer()
 ans  =

    0.74

-->timer(); y3 = phi3(t,a,b,c); timer()
 ans  =

    0.68
```

Remarque : pour ce type de calcul, la fonction `find` est plus naturelle et plus simple à utiliser : sur un vecteur de booléens `b` elle renvoie un vecteur contenant les indices `i` tels que `b(i)=%t` (la matrice vide si toutes les composantes sont fausses). Exemple :

```
-->x = rand(1,6)
 x  =
!   0.8497452    0.6857310    0.8782165    0.0683740    0.5608486    0.6623569 !

-->ind = find( 0.3<x & x<0.7 )
 ind  =
!   2.    5.    6. !
```

16. dans lesquels la fonction `bool2s` permet de convertir une matrice de booléens en matrice de réels (vrai donnant 1 et faux 0)

Sur une matrice booléenne `A` vous obtenez la même liste en considérant que la matrice est un « grand » vecteur où les éléments de `A` ont été réordonnés « colonne par colonne ». Il est cependant possible avec un deuxième argument de sortie de récupérer la liste des indices de ligne et de colonne :

```
-->A = rand(2,2)
 A   =
!   0.7263507    0.5442573 !
!   0.1985144    0.2320748 !

-->[il,ic]=find(A<0.5)
 ic  = ! 1.    2. !
 il  = ! 2.    2. !
```

Voici maintenant un code pour la fonction ϕ qui utilise `find` :

```
function [y]=phi4(t,a,b,c)
  // on utilise la fonction find plus naturelle
  t_le_b = ( t <= b )
  indices_a_b = find( a<t & t_le_b )
  indices_b_c = find( ~t_le_b & t<c )
  y = zeros(t)
  y(indices_a_b) = (t(indices_a_b) - a)/(b - a)
  y(indices_b_c) = (c - t(indices_b_c))/(c - b)
```

Conclusion : si vos calculs commencent à être trop longs, essayer de les « vectoriser ». Si cette vectorisation est impossible ou insuffisante il ne reste plus qu'à écrire les parties cruciales en C ou en fortran 77.

3.6 Exercices

1. Écrire une fonction pour résoudre un système linéaire où la matrice est triangulaire supérieure. On pourra utiliser l'instruction `size` qui permet de récupérer les deux dimensions d'une matrice :
   `-->[n,m]=size(A)`
   Dans un premier temps, on programmera l'algorithme classique utilisant deux boucles, puis on essaiera de remplacer la boucle interne par une instruction matricielle. Pour tester votre fonction, vous pourrez générer une matrice de nombres pseudo-aléatoires et n'en garder que la partie triangulaire supérieure avec l'instruction `triu` :
   `-->A=triu(rand(4,4))`
2. La solution du système d'équations différentielles du 1$^{\text{er}}$ ordre :
   $$\frac{dx}{dt}(t) = Ax(t), \quad x(0) = x_0 \in \mathbb{R}^n, \quad x(t) \in \mathbb{R}^n, \quad A \in \mathcal{M}_{nn}(\mathbb{R})$$
   peut être obtenue en utilisant l'exponentielle de matrice (cf votre cours d'analyse de 1$^{\text{ère}}$ année) :
   $$x(t) = e^{At}x_0$$
   Comme Scilab dispose d'une fonction qui calcule l'exponentielle de matrice (`expm`), il y a sans doute quelque chose à faire. On désire obtenir la solution pour $t \in [0,T]$. Pour cela, on peut la calculer en un nombre n suffisamment grand d'instants uniformément répartis dans cet intervalle $t_k = k\delta t, \quad \delta t = T/n$ et l'on peut utiliser les propriétés de l'exponentielle pour alléger les calculs :
   $$x(t_k) = e^{Ak\delta t}x_0 = e^{k(A\delta t)}x_0 = (e^{A\delta t})^k x_0 = e^{A\delta t}x(t_{k-1})$$

ainsi il suffit uniquement de calculer l'exponentielle de la matrice $A\delta t$ puis de faire n multiplications « matrice vecteur » pour obtenir $x(t_1)$,$x(t_2)$, ... ,$x(t_n)$. Écrire un script pour résoudre l'équation différentielle (un oscillateur avec amortissement) :

$$x'' + \alpha x' + kx = 0, \text{ avec par exemple } \alpha = 0.1, k = 1, x(0) = x'(0) = 1$$

que l'on mettra évidemment sous la forme d'un système de deux équations du premier ordre. À la fin on pourra visualiser la variation de x en fonction du temps, puis la trajectoire dans le plan de phase. On peut passer d'une fenêtre graphique à une autre avec l'instruction `xset("window",window-number)`. Par exemple :

```
--> //fin des calculs
--> xset(''window'',0) // on selectionne la fenetre numero 0
--> instruction pour le premier graphe (qui  s'affichera sur la fenetre 0)
--> xset(''window'',1) // on selectionne la fenetre numero 1
--> instruction pour le deuxieme graphe (qui s'affichera sur la fenetre 1)
```

3. Écrire une fonction `[i,info]=intervalle_de(t,x)` pour déterminer l'intervalle i tel que $x_i \leq t \leq x_{i+1}$ par la méthode de la dichotomie (les composantes du vecteur x étant telles que $x_i < x_{i+1}$). Si $t \notin [x_1,x_n]$, la variable booléenne `info` devra être égale à `%f` (et `%t` dans le cas inverse).
4. Récrire la fonction `myhorner` pour quelle s'adapte au cas où l'argument `t` est une matrice (la fonction devant renvoyer une matrice `p` (de même taille que `t`) où chaque coefficient (i,j) correspond à l'évaluation du polynôme en `t(i,j)`).
5. Écrire une fonction `[y] = signal_fourier(t,T,cs)` qui renvoie un début de série de Fourier en utilisant les fonctions :

$$f_1(t,T) = 1,\ f_2(t,T) = \sin(\frac{2\pi t}{T}t),\ f_3(t,T) = \cos(\frac{2\pi t}{T}), f_4(t,T) = \sin(\frac{4\pi t}{T}),\ f_5(t,T) = \cos(\frac{4\pi t}{T}), \cdots$$

au lieu des exponentielles. T est un paramètre (la période) et le signal sera caractérisé (en dehors de sa période) par le vecteur `cs` de ces composantes dans la base $f_1,f_2,f_3,\cdots$. On récupèrera le nombre de fonctions à utiliser à l'aide de l'instruction `length` appliquée sur `cs`. Il est recommandé d'utiliser une fonction auxiliaire `[y]=f(t,T,k)` pour calculer $f_k(t,T)$. Enfin tout cela doit pouvoir s'appliquer sur un vecteur (ou une matrice) d'instants `t`, ce qui permettra de visualiser facilement un tel signal :

```
--> T = 1 // une periode ...
--> t = linspace(0,T,101) // les instants ...
--> cs =  [0.1 1 0.2 0 0  0.1] // un signal avec  une composante continue
--> // du fondamental, pas d'harmonique  1 (periode 2T) mais une harmonique 2
--> [y] = signal_fourier(t,T,cs); // calcul du signal
--> plot(t,y) // et un dessin ...
```

6. Voici une fonction pour calculer le produit vectoriel de deux vecteurs :

```
function [v]=prod_vect(v1,v2)
  // produit vectoriel v = v1 /\ v2
  v(1) = v1(2)*v2(3) - v1(3)*v2(2)
  v(2) = v1(3)*v2(1) - v1(1)*v2(3)
  v(3) = v1(1)*v2(2) - v1(2)*v2(1)
```

Vectoriser ce code de manière à calculer dans une même fonction `function [v]=prod_vect_v(v1,v2)` les produits vectoriels $v^i = v_1^i \wedge v_2^i$ où v^i, v_1^i et v_2^i désigne la i$^{\text{ème}}$ colonne des matrices (3,n) contenant ces vecteurs.

Chapitre 4

Les graphiques

Dans ce domaine, Scilab possède de nombreuses possibilités qui vont de primitives de bas niveau[1], à des fonctions plus complètes qui permettent en une seule instruction de tracer toutes sortes de graphiques types. Dans la suite, j'explique seulement une petite partie de ces possibilités. *Remarque :* pour ceux qui connaissent les instructions graphiques de MATLAB[2], Stéphane Mottelet a écrit une bibliothèque de fonctions Scilab pour faire des « plots » à la MATLAB ; ça se récupère là :
`http://www.dma.utc.fr/~mottelet/scilab/`

4.1 Les fenêtres graphiques

Lorsque l'on lance une instruction comme `plot, plot2d, plot3d ...` alors qu'aucune fenêtre graphique n'est activée, Scilab choisit de mettre le dessin dans la fenêtre de numéro 0. Si vous relancez un tel graphe, il va alors en général s'afficher par dessus le premier[3], et il faut auparavant, effacer la fenêtre graphique, ce qui peut se faire soit à partir du menu de cette fenêtre (item `clear` du menu `File`), soit à partir de la fenêtre Scilab avec l'instruction `xbasc()`. En fait on peut jongler très facilement avec plusieurs fenêtres graphiques à l'aide des instructions suivantes :

`xset("window",num)`	la fenêtre courante devient la fenêtre de numéro `num` ; si cette fenêtre n'existait pas, elle est créée par Scilab.
`xselect()`	met en « avant » la fenêtre courante ; si aucune fenêtre graphique n'existe, Scilab en crée une.
`xbasc([num])`	efface la fenêtre graphique numéro `num` ; si `num` est omis, Scilab efface la fenêtre courante.
`xdel([num])`	détruit la fenêtre graphique numéro `num` ; si `num` est omis, Scilab détruit la fenêtre courante.

D'une manière générale, lorsque l'on a sélectionné la fenêtre courante (par `xset("window",num)`) , la famille d'instructions `xset("nom",a1,a2,...)` permet de régler tous les paramètres de cette fenêtre : `"nom"` désigne généralement le type de paramètre à ajuster comme `font` pour la police de caractère utilisée (pour le titre et les légendes diverses), `"thickness"` pour l'épaisseur des traits, `"colormap"` pour la carte des couleurs, etc, suivi d'un ou plusieurs arguments pour le réglage proprement dit. L'ensemble de ces paramètres forme ce que l'on appelle le contexte graphique (chaque fenêtre peut donc avoir son propre contexte graphique). Pour le détail sur ces paramètres (assez nombreux) voir le `Help` à la rubrique `Graphic Library` mais la plupart d'entre-eux peuvent se régler interactivement par un menu graphique qui apparaît suite à la commande `xset()` (remarque : ce menu affiche aussi la carte des couleurs (mais il ne permet pas de la modifier)). Enfin la famille d'instructions `[a1,a2,...]=xget('nom')` permet de récupérer les divers paramètres du contexte graphique.

1. exemples : tracés de rectangles, de polygones (avec ou sans remplissage), récupérer les coordonnées du pointeur de la souris
2. généralement plus simples que celles de Scilab !
3. sauf avec `plot` qui efface automatiquement le contenu de la fenêtre courante

4.2 Les courbes dans le plan

4.2.1 Introduction à plot2d

Nous avons déjà vu l'instruction `plot` toute simple. Cependant si l'on veut dessiner plusieurs courbes mieux vaut se concentrer sur `plot2d`. L'utilisation la plus basique est la suivante :

```
-->x=linspace(-1,1,61)'; // les abscisses (en vecteur colonne)

-->y = x.^2; // les ordonnees (aussi en vecteur colonne)

-->plot2d(x,y) // --> il lui faut des vecteurs colonnes !
```

rajoutons maintenant une autre courbe :

```
-->ybis = 1 - x.^2;

-->plot2d(x,ybis)

-->xtitle("Courbes...")  // je rajoute un titre
```

ici tout se passe bien car Scilab choisit la même échelle, mais si on continue avec :

```
-->yter = 2*y;

-->plot2d(x,yter)
```

on remarque que Scilab change l'échelle (cf les graduations des ordonnées) et cette nouvelle courbe se confond avec la première (mais il est possible de garder l'ancienne échelle…) Affichons maintenant les 3 courbes simultannément :

```
-->xbasc() // pour effacer

-->plot2d([x x x],[y ybis yter]) // concatenation de matrices ...

-->xtitle("Courbes...","x","y") // un titre plus une legende pour les deux axes
```

et vous devez obtenir quelque chose qui ressemble à la figure 4.1.

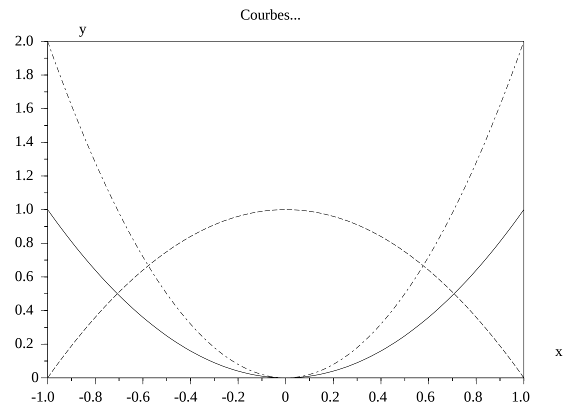


FIG. 4.1 – *Les fonctions* x^2, $1 - x^2$ *et* $2x^2$

Ainsi pour afficher simultannément plusieurs courbes, l'instruction prend la forme `plot2d(Mx,My)` où `Mx` et `My` sont deux matrices de tailles identiques, le nombre de courbes étant égal au nombre de colonnes

`nc`, et la i$^{\text{ème}}$ courbe est obtenue à partir des vecteurs `Mx(:,i)` (ses abscisses) et `My(:,i)` (ses ordonnées). Passons maintenant à la syntaxe générale :

`plot2d(Mx,My,[style,strf,leg,rect,nax])`

où les arguments optionnels ont la signification suivante :

- `style` est un vecteur ligne de dimension `nc`, sa i$^{\text{ème}}$ composante spécifiant le style de la i$^{\text{ème}}$ courbe. Sur un terminal couleur, si `style(i)` est un entier strictement positif k, la courbe correspondante sera dessinée avec la couleur numéro k (pour quelques unes des couleurs de la carte (cf table 4.1 mais `xset()` vous les montrera toutes) alors que sur un terminal noir et blanc, on aura des traits avec différents pointillés[4]. Par contre si `style(i)` est un entier négatif ou nul, chaque point de la courbe est repéré par un petit symbole (point, triangles, ronds, croix, plus, etc ...) (les points ne sont alors pas reliés) dont vous trouverez la liste complète sur la figure 4.2.
- `strf` est une chaîne de trois caractères `"xyz"` telle que :
  - si `x` est le caractère `1` alors une légende pour chacune des courbes (donnée par `leg` qui doit alors être présent dans la liste des arguments) sera affichée ; pour les autres valeurs de `x` la légende n'est pas affichée ;
  - le caractère `y` spécifie le calcul de l'échelle (c-à-d le rectangle x_{min},y_{min},x_{max},y_{max} qui va déterminer la frontière du graphe) :
    - pour `y = 0` on utilise l'échelle précédente (celle qui a été déterminée par un appel précédent) ;
    - pour `y = 1` l'échelle est donnée par l'argument `rect` ;
    - pour `y = 2` Scilab calcule automatiquement l'échelle (en prenant les max et min de `Mx` et `My`) ;
    - avec `y = 3` et `y = 4` on peut obtenir une échelle isométrique (voir plus loin) ;
    - pour les autres possibilités voir le `Help`
  - le caractère `z` permet de régler le pourtour du graphe :
    - pour `z=1` le graphe est entouré d'une boite (dont les dimensions sont x_{min},y_{min},x_{max},y_{max}) et les côtés ouest (l'axe des y) et sud (l'axe des x) de la boite sont gradués (la graduation est automatique ou est spécifiée par le paramètre `nax`) ;
    - pour `z=2` le graphe est simplement entouré de la boite (x_{min},y_{min},x_{max},y_{max}) ;
    - pour les autres valeurs, il n'y a ni boite, ni axes.
- Le paramètre `leg` est une chaîne de caractère qui donne une légende (cf `x=1`) pour chacune des courbes `'y1@y2@...'`, chaque légende étant séparée de la suivante par le caractère `@` ; si vous fournissez n légendes alors qu'il y a nc courbes ($n < nc$) seules les n premières courbes auront une légende ;
- `rect = [xmin,ymin,xmax,ymax]` sert à spécifier l'échelle (cf `y=1`) ;
- `nax = [nx,Nx,ny,Ny]` sert à choisir les graduations (cf `z=1`) ; `Nx` étant le nombre d'intervalles en « x » (on obtient alors `Nx+1` valeurs numériques), `nx` étant le nombre de sous-intervalles par intervalle (ainsi une valeur de `nx=2` donnera une petite graduation entre deux grandes).

Attention : si vous avez besoin du i$^{\text{ème}}$ argument, il faut impérativement fournir tous les arguments précédents. Ainsi, si vous vous voulez choisir précisement les graduations (avec `z= 1` et `nax`), il faut alors fournir tous les paramètres précédents (avec éventuellement des valeurs sans signification si elles ne sont pas utilisées : par exemple `leg=' ',rect=[1:4]`).

Voici un premier exemple avec presque tous les paramètres (cf figure 4.3) :

```
-->t = linspace(0,2*%pi,60)';

-->x1 = 2*cos(t); y1 = sin(t); // pour une ellipse

-->x2 = cos(t); y2 = y1; // pour un cercle

-->x3 = linspace(-2,2,60)'; y3 = erf(x3); // la fct erreur
```

4. sur un terminal couleur l'instruction `xset("use color",0)` permet de passer en mode noir et blanc, et `xset("use color",1)` permet de revenir en mode couleur

numéro	couleur
1	noir
2	un bleu foncé
3	un vert clair
4	un bleu clair (cyan?)
5	un rouge vif
6	un mauve
26	un marron
29	un rose
32	jaune orangé

TAB. 4.1 – *Quelques couleurs de la carte par défaut*

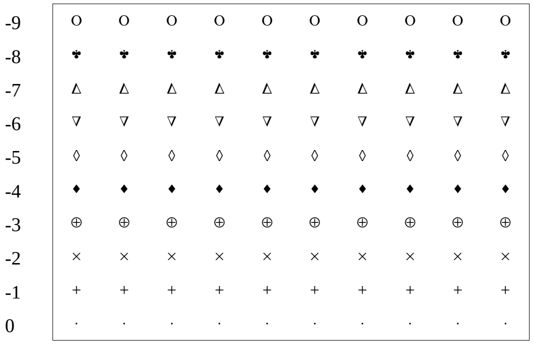


FIG. 4.2 – *Les différents marqueurs*

```
-->rect = [-2.05,-1.4,2.05,1.4]; // la fenetre

-->leg="ellipse@cercle@fct erreur"; // les legendes

-->plot2d([x1 x2 x3],[y1 y2 y3],[1:3],"111",leg,rect)

-->xtitle("Encore des courbes ...","x","y")
```

D'une manière générale, on utilisera l'instruction `plot2d` de la façon suivante (où `nc` est le nombre de courbes) :

```
plot2d(Mx,My,[1:nc],"121","leg_1@....@leg_nc")
```

qui permet de repérer chacune des différentes courbes par une légende, tout en calculant automatiquement l'échelle[5]. Par contre imposer la fenêtre `rect` avec `y=1` peut être utile :

– soit pour faire joli, car avec le dimensionnement automatique les courbes collent à la fenêtre ; on peut alors la calculer par :

```
x0 = min(Mx); x1 = max(Mx);
y0 = min(My); y1 = max(My);
dx = (x1 - x0)*0.05; dy= (y1 - y0)*0.05;
// et enfin :
xmin = x0 - dx; xmax = x1 + dx; ymin  = y0 - dy; ymax = y1 + dy;
```

5. l'instruction simple `plot2d(Mx,My)` correspond en fait à `plot2d(Mx,My,[1:nc],"021")`

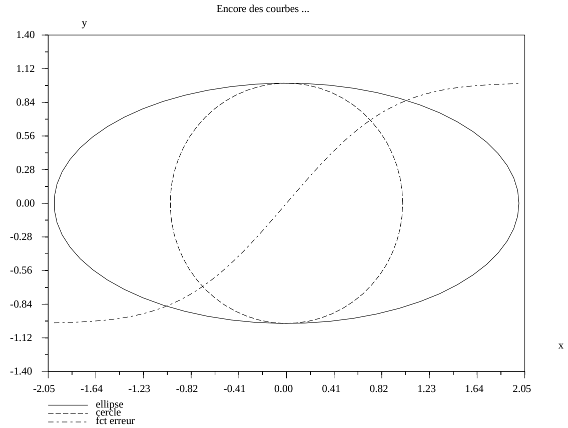


FIG. 4.3 – *Ellipse, cercle et erf*

– ou alors lorsque que l'on veut afficher une suite de courbes où chaque i$^{\text{ème}}$ courbe est obtenue à la fin de la i$^{\text{ème}}$ étape d'un calcul (chaque courbe étant affichée dès qu'elle est disponible, au lieu d'attendre la fin de tous les calculs). Voici un tel exemple :

```
x = linspace(-1,1,101)';
rect = [-1,-1.05,1,1.05];   // la fenetre de visualisation
t=linspace(2,-2,7)
y = t(1)*x.^2;  // un 1er calcul
xbasc()         // nettoyage de la fenetre graphique courante
xselect()       // mise en  avant de la fenetre graphique courante
xtitle("Des courbes...","x","y")
plot2d(x,y,1,"011"," ",rect)
// affichage
for i=2:7
  y = t(i)*x.^2;      // calcul numero i
  xpause(2000)        // pour simuler un calcul plus long que le precedent...
  plot2d(x,y,i,"000") // on utilise l'echelle precedente (la fenetre de visu)
end
```

Pour simuler des calculs plus longs, j'ai utilisé la fonction `xpause(dt)` qui permet de figer l'affichage pendant environ un temps de `dt` ms (ceci dit cette fonction est très dépendante du système et ne fonctionne pas toujours).

4.2.2 Dessiner plusieurs courbes qui n'ont pas le même nombre de points

Avec `plot2d` on ne peut pas dessiner plusieurs courbes qui n'ont pas été discrétisées avec le même nombre d'intervalles. On est alors obligé de calculer l'échelle[6] puis d'appeler `plot2d` une première fois en imposant l'échelle (avec `y=1`), les autres appels devant utiliser l'échelle précédente `y=0`. Voici un exemple sous la forme d'un script (cf figure 4.4):

```
x1 = linspace(0,1,61)';
x2 = linspace(0,1,31)';
x3 = linspace(0.1,0.9,12)';
y1 = x1.*(1-x1).*cos(2*%pi*x1);
y2 = x2.*(1-x2);
```

6. ou alors on peut envoyer en premier une courbe qui déterminera l'échelle (`y=2`)

```
y3 = x3.*(1-x3) + 0.1*(rand(x3)-0.5);  // idem a y2 avec une perturbation
ymin = min(min(y1),min(y2),min(y3));
ymax = max(max(y1),max(y2),max(y3));
dy = (ymax - ymin)*0.05;
rect = [0,ymin - dy,1,ymax+dy];    // fenetre de visualisation
xbasc()                            // effacement des graphiques precedents...
plot2d(x1,y1,1,"011"," ",rect)     // 1er appel qui impose la fenetre de visu
plot2d(x2,y2,2,"000")              // 2eme
plot2d(x3,y3,-1,"000")             // et 3 eme appel : on utilise l'echelle precedente
xtitle("Des courbes...","x","y")
```

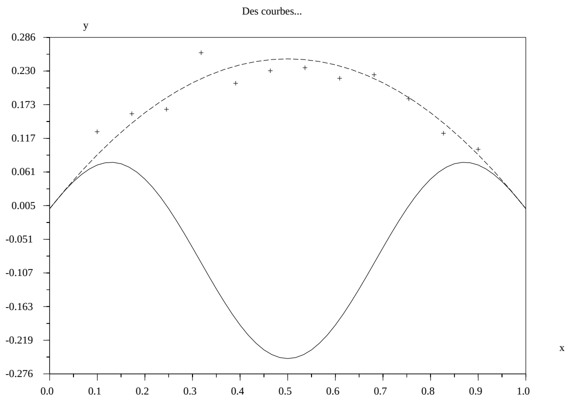


FIG. 4.4 – *Encore des courbes...*

On ne peut pas alors avoir une légende pour chacune des courbes, mais cela reste possible en programmant avec les fonctions graphiques de bas niveau. Pour « customiser » ses graphes la bonne solution est d'exporter en format fig puis d'utiliser le logiciel de dessin vectoriel xfig (cf Récuperer ses graphiques... un peu plus loin).

4.2.3 Dessiner en utilisant une échelle isométrique

Par défaut les graphiques ne sont pas tracés avec une échelle isométrique et si vous essayez de dessiner un cercle, il va généralement ressembler plutôt à une ellipse. En retaillant la fenêtre graphique on peut arranger ce problème mais les sorties en postscript et autres utiliseront l´échelle initiale. Pour obtenir une échelle isométrique il existe plusieurs possibilités :

1. lors du premier appel à `plot2d`, le paramètre `y` de la chaîne `strf` doit être égal à 3 et on explicite le rectangle de visualisation avec l'argument `rect`, ou bien on choisit `y = 4` et le rectangle de visualisation est calculé automatiquement (à partir des minima et maxima des points) ; pour les appels suivants à `plot2d`, le paramètre `y` doit être égal à 0 ;
2. avant les appels à `plot2d`, on règle l'échelle avec l´instruction `isoview` :

   `isoview(xmin, xmax, ymin, ymax)`

   et lors de(s) appel(s) à `plot2d`, il faut impérativement mettre le paramètre `y` (de la chaîne `strf`) à 0.

Ces deux méthodes peuvent s'utiliser avec la plupart des primitives de dessins 2d. L'échelle isométrique est maintenue lorsque l'on retaille la fenêtre ainsi que lorsque l'on grossit une partie avec le « zoom » interactif. Voici un exemple avec `isoview` (cf figure 4.5) :

```
// un exemple d'utilisation d'isoview
```



```
// 1/ les donnees pour tracer un cercle
t = linspace(0,2*%pi,100)';
x = cos(t) ; y = sin(t) ;
// 2/ les donnees pour dessiner des points << pres >> du cercle
tt = linspace(0,2*%pi,40)' ;
xx = cos(tt) + 0.2*(rand(tt) - 0.5);
yy = sin(tt) + 0.2*(rand(tt) - 0.5);
// 3/ c'est parti pour le dessin
xbasc()                         // effacement (au cas ou)
isoview(-1.2, 1.2, -1.2, 1.2)   // reglage de l'echelle
xset("font",3,1)                // selectionne time-italic en 10 pts
plot2d(x,y,2,"001")             // dessin du cercle
plot2d(xx,yy,-9,"000")          // dessin des points
xset("font",4,2)                // selectionne time-bold en 12 pts
xtitle("Un exemple d''utilisation d''isoview : essayer de retailler la fenetre","x","y")
xselect()                       // mise en avant de la fenetre graphique
```

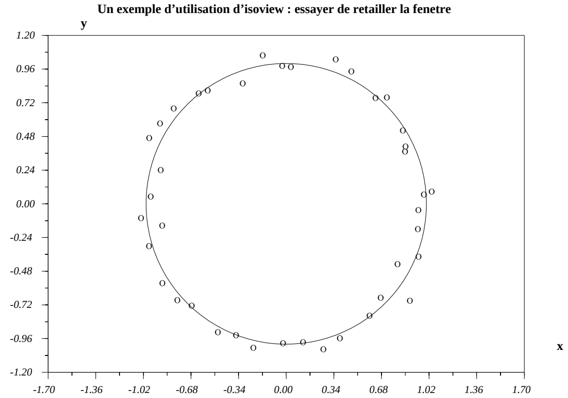


FIG. 4.5 – *Un cercle qui ressemble à un cercle avec des points autour...*

Avec la première méthode, la fin du script précédent s'écrirait :

```
// 3/ c'est parti pour le dessin
xbasc()                         // effacement (au cas ou)
xset("font",3,1)                // selectionne time-italic en 10 pts
plot2d(x,y,2,"031"," ",[-1.2, -1.2, 1.2, 1.2])   // dessin du cercle
plot2d(xx,yy,-9,"000")          // dessin des points
xset("font",4,2)                // selectionne time-bold en 12 pts
xtitle("Echelle isometrique : essayer de retailler la fenetre","x","y")
xselect()                       // mise en avant de la fenetre graphique
```

4.3 Récupérer ses graphiques sous plusieurs formats

Ceci est très facile à partir du menu `File` de la fenêtre graphique, en choisissant l'item `Export`, un autre menu vous propose différents choix tournant autour du langage postscript ainsi que le format fig qui permet lui de retravailler son graphique avec le logiciel de dessin vectoriel xfig. Depuis la version 2.5 vous pouvez aussi exporter en gif.

4.4 Animations simples

Il est facile de réaliser des petites animations avec Scilab qui permet l'utilisation de la technique du double tampon (double buffer) évitant les scintillements ainsi que celle des masques pour faire évoluer un objet sur un fond fixe. D'autre part il y a deux « drivers » différents qui permettent l'affichage à l'écran [7], l'un `Rec` qui enregistre toutes les opérations graphiques effectuées dans la fenêtre et qui est le driver par défaut, l'autre `X11` qui se contente simplement d'afficher les graphiques (il est alors impossible de « zoomer »). Pour une animation qui comporte beaucoup de dessins il est préférable d'utiliser ce dernier ce qui se fait par l'instruction `driver("X11")` (et `driver("Rec")` pour revenir au driver par défaut) . Par contre si vous restez en `Rec` un redimensionnement de la fenêtre graphique vous permettra de revoir votre animation. Voici un petit script qui montre l'intérêt d'utiliser la technique du double tampon. On affiche au fur et à mesure du temps (avec une discrétisation...), la fonction :

$$y(x,t) = \cos(t - 2x),\ x \in [0,2\pi],\ \text{ et } t \in [0,6\pi].$$

Pour passer d'un dessin à l'autre j'efface une partie de la fenêtre avec `xclea`. Vous pouvez simplifier en utilisant `xset("wwpc")` qui nettoie complètement le buffer courant (il faut alors récrire le titre, et redessiner le contour et les axes du graphe à chaque fois). Pour des animations plus subtiles vous devez jouer avec les masques en utilisant `xset('alufunction',num)`, où `num` est un entier précisant la fonction logique d'affichage, cf le Help et les démos... en attendant que je complète cette partie !

J'espère que ce script est suffisamment commenté pour que vous puissiez le comprendre (c'est en tout cas un bon exercice).

```
// script pour tester une animation simple

// 0) initialisations diverses

nt =  201;  nx =  101;
T = linspace(0,6*%pi,nt);  // les instants
x = linspace(0,2*%pi,nx)'; // les abscisses
rect  = [-0.05,-1.05,2*%pi+0.05,1.05]; // la fenetre
xselect() //  mise   en  avant de la   fenetre   graphique courante
xbasc()   //  effacement de cette fenetre (au cas ou)

// 1) 1ere methode (la mauvaise) : on utilise pas le double tampon

// titlepage permet d'afficher un titre au milieu de la fenetre graphique
titlepage(["Essais  d''animation";"  ";"methode  1   :";"affichage direct"])
xtitle("(cliquer dans la fenetre pour lancer l''animation)")
[c_i,c_x,c_y,c_w]=xclick(); // permet d'attendre un clic souris ...
xbasc() //  on  efface le titre
xtitle("Une  onde    progressive","x","y")
y =  cos(T(1) - 2*x);
plot2d(x,y,1,"112","y(x,t)=cos(t-2x)",rect)
for i=2:nt
   xclea(-0.02,1.02,2*%pi+0.04,2.04) // permet d'effacer un rectangle
   y = cos(T(i) - 2*x);
   plot2d(x,y,1,"000")  // affichage   de la  courbe  seule  (sans boite  ni
end                     // legende, mais comme on ne les a pas effacees...)

// 2) 2 eme methode (la bonne) : on utilise le double buffer

xbasc()
titlepage(["Essais  d''animation";" ";"methode  2 :";"affichage par double buffer"])
xtitle("(cliquer dans la fenetre pour lancer l''animation)")
[c_i,c_x,c_y,c_w]=xclick();
xbasc() // on efface le deuxieme titre
xset("pixmap",1)  // pour utiliser le double buffer (les dessins ne sont pas envoyes
                  // directement a l'ecran mais dans un buffer video (pixmap)
xtitle("Une onde progressive","x","y")
y = cos(T(1) - 2*x);
plot2d(x,y,1,"112","y(x,t)=cos(t-2x)",rect) // envoi dans le buffer non affiche a l'ecran
```

7. en plus des drivers qui permettent de faire des dessins en postscript et en fig


```
xset("wshow")                                   // affichage : bascule entre les 2 buffers
for i=2:nt
   xclea(-0.02,1.02,2*%pi+0.04,2.04)
   y = cos(T(i) - 2*x);
   plot2d(x,y,1,"000")  // envoi dans le buffer non affiche a l'ecran
   xset("wshow")        // affichage : bascule entre les 2 buffers
end
xset("pixmap",0) // on  revient dans le mode ou les graphiques
                 // sont directement affiches a l'ecran
```

4.5 Les surfaces

L'instruction générique pour dessiner des surfaces est `plot3d`. `plot3d1` qui s'utilise de façon quasi identique et permet de rajouter des couleurs selon la valeur en z. Avec une représentation de votre surface par facettes, ces 2 instructions permettent aussi d'avoir une couleur différente pour chaque facette.

4.5.1 Introduction à plot3d

Si votre surface est donnée par une équation du type $z = f(x,y)$, il est particulièrement simple de la représenter pour un domaine rectangulaire des paramètres. Dans l'exemple qui suit je représente la fonction $f(x,y) = cos(x)cos(y)$ pour $(x,y) \in [0,2\pi] \times [0,2\pi]$:

```
-->x=linspace(0,2*%pi,31);   // la discretisation en x (et aussi en y : c'est la meme)

-->z=cos(x)'*cos(x);  // le jeu des valeurs en z : une matrice z(i,j) = f(x(i),y(j))

-->plot3d(x,x,z)  // le dessin
```

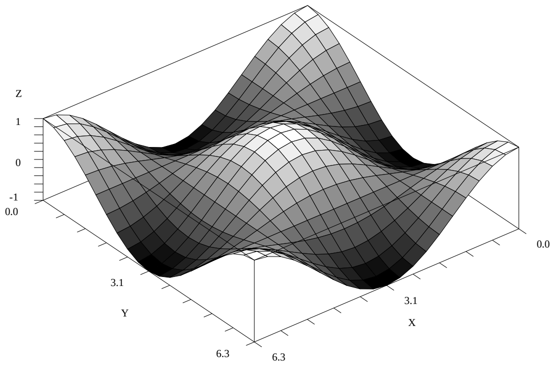


FIG. 4.6 – *la fonction* $z = cos(x)cos(y)$

Vous devez alors obtenir quelque chose qui ressemble à la figure (4.6) [8]. Ici j'ai utilisé `plot3d` dans sa syntaxe la plus simple mais vous pouvez préciser le point de vue et un certain nombre d'autres paramètres :

```
plot3d(x,y,z [,theta,alpha,leg [,flag,ebox]])
plot3d1(x,y,z [,theta,alpha,leg [,flag,ebox]])
```

où :

1. `x` et `y` sont deux vecteurs lignes ($(1,nx)$ et $(1,ny)$) correspondants à la discrétisation en x et en y ;

8. sauf que j'ai utilisé des couleurs avec `plot3d1` (transformées en niveau de gris pour ce document) et le point de vue est un peu différent

2. `z` est une matrice ((nx,ny)) telle que `z(i,j)` est « l'altitude » au point `(x(i),y(j))` ;
3. `theta` et `alpha` sont les deux angles (en degré) précisant le point de vue ;
4. `leg` est une chaîne de caractères pour obtenir une légende pour chacun des axes (par exemple `leg="x@y@z"`) ;
5. `flag` est un vecteur à trois composantes `flag=[mode type box]` où :
   (a) le paramètre `mode` spécifie si on dessine ou non les faces cachées : si `mode >0` alors les faces cachées sont enlevées[9] , si `mode = 0` c'est le contraire et si `mode <0` les faces cachées sont enlevées de même que le maillage (à partir de la version 2.3.1) ;
   (b) si `type = 0` on utilise l'échelle précédente, si `type = 1` on précise l'échelle avec :
   `ebox = [xmin,xmax,ymin,ymax,zmin,zmax]`
   sinon pour tout autre valeur l'échelle est calculée automatiquement avec les données ;
   (c) le paramètre `box` contrôle le pourtour du graphe :
   - `box = 0` rien n'est dessiné autour,
   - `box = 2` les axes sont dessinés,
   - `box = 3` une boite entoure la surface...,
   - `box = 4` une boite et les axes...

Voici un petit script où on utilise presque tous les paramètres de `plot3d`. C'est une animation qui vous permettra de comprendre le changement de point de vue avec les paramètres `theta` et `alpha` :

```
x=linspace(-%pi,%pi,31);
z=sin(x)'*sin(x);
n = 21
theta = linspace(20,80,n);
xbasc(); xselect()
xset("pixmap",1)   // pour activer le double buffer
alpha = linspace(60,30,n);
plot3d(x,x,z,theta(1),alpha(1),"x@y@z",[2 2 4])
xtitle("variation du point de vue avec le parametre theta")
xset("wshow")
// on fait varier theta
for i=2:n
  xset("wwpc") // effacement du buffer courant
  plot3d(x,x,z,theta(i),alpha(1),"x@y@z",[2 0 4])
  xtitle("variation du point de vue avec le parametre theta")
  xset("wshow")
end
// on fait varier alpha
for i=2:n
  xset("wwpc") // effacement du buffer courant
  plot3d(x,x,z,theta(n),alpha(i),"x@y@z",[2 0 4])
  xtitle("variation du point de vue avec le parametre alpha")
  xset("wshow")
end
xset("pixmap",0)  // on revient dans le mode initial
```

4.5.2 La couleur

Vous pouvez réessayer les deux exemples précédents en remplaçant `plot3d` par `plot3d1` qui met des couleurs selon la valeur en z. Votre surface va alors ressembler à une mosaïque car la carte des couleurs par défaut n'est pas « continue ». À partir de la version 2.4 la fonction `hotcolormap` fournit une telle carte. Une carte des couleurs est une matrice de dimensions `(nb_couleurs,3)`, la $i^{\text{ème}}$ ligne correspondant à l'intensité (comprise entre 0 et 1) en rouge, vert et bleu de la $i^{\text{ème}}$ couleur. Étant donné

9. prendre une valeur de 2, mode=1 produit des effets bizarres... du moins sur ma machine

une telle matrice que nous appellerons `C`, l'instruction `xset("colormap",C)` permet de la charger dans le contexte graphique de la fenêtre graphique courante. Petite remarque : si vous changez la carte des couleurs après avoir dessiné un graphique, les changements ne se répercutent pas immédiatement sur votre dessin (ce qui est normal). Il suffit par exemple de retailler la fenêtre graphique ou alors d'envoyer l'ordre `xbasr(numero_fenetre)` pour redessiner (et la nouvelle carte est utilisée). Voici une petite fonction qui installe automatiquement une carte des couleurs très classique. Elle utilise un nombre de couleurs multiple de huit mais elle se débrouille si ce n'est pas le cas. D'autre part vous pouvez l'appeler sans paramètre et elle installe une carte avec 64 couleurs.

```
function carte_couleurs(nb_couleurs)
  // installe une carte de couleurs classique avec nb_couleurs si
  // nb_couleurs est un multiple de huit. Sinon on prend le multiple
  // de huit le plus proche en valeur sup. Si l'argument nb_couleurs
  // est absent on impose nb_couleurs=64; on limite aussi le nb_couleurs
  // entre 8 et 128
  [lhs,rhs]=argn(0);
  if rhs == 0 then
    n = 8 ; nb_couleurs = 64
  else
    n = ceil(nb_couleurs/8)
    if (n < 1) then, n = 1, end      // il faut une carte... donc au moins 8 couleurs
    if (n > 16) then, n = 16, end    // limitation a 128 couleurs
    if (8*n ~= nb_couleurs) then
       nb_couleurs = 8*n
       warning(" La carte aura " + sprintf("%d",nb_couleurs) + " couleurs")
    end
  end
  rgb = zeros(nb_couleurs,3)
  // le vecteur qu'il suffit de decaler ensuite :
  sequence = [(1:2*n)'/(2*n) ; ones(2*n,1); (2*n:-1:1)'/(2*n)]
  rgb(3*n+1:$,1) = sequence(1:5*n)
  rgb(n+1:7*n,2) = sequence
  rgb(1:5*n  ,3) = sequence(n+1:$)
  // et il n'y a plus qu'a installer cette carte dans le contexte graphique
  xset("colormap",rgb)
```

4.5.3 `plot3d` et `plot3d1` avec des facettes

Pour utiliser ces fonctions dans un contexte plus général, il faut donner une description de votre surface par facettes. Celle-ci est constituée par 3 matrices `xf, yf, zf` de dimensions `(nb_sommets_par_face, nb_faces)` où `xf(j,i),yf(j,i),zf(j,i)` sont les coordonnées du $j^{\text{ème}}$ sommets de la $i^{\text{ème}}$ facette. Modulo ce petit changement, ces fonctions s'utilisent comme précédemment pour les autres arguments :

```
plot3d(xf,yf,zf [,theta,alpha,leg [,flag,ebox]])
plot3d1(xf,yf,zf [,theta,alpha,leg [,flag,ebox]])
```

D'autre part si vous voulez des couleurs, il faut donner la suite des sommets d'une facette dans le sens de la normale intérieure pour le tire-bouchon. Et si vous voulez imposer des couleurs pour chaque facette, le troisième argument doit être une liste : `list(zf,colors)` où `colors` est un vecteur de taille `nb_faces`, `colors(i)` donnant le numéro de la couleur (dans la carte) attribué à la $i^{\text{ème}}$ facette.

Reprenons l'exemple du cube donné dans la section sur les listes typées. La description donnée n'est pas celle attendue par `plot3d` mais une petite fonction va permettre de l'obtenir :

```
function [xf,yf,zf] = trans_polyedre(Polyedre)
  // cette fonction permet d'obtenir la description par facettes
  // attendue par plot3d a partir d'une description classique
  [ nb_faces, nb_s_par_face ] = size(Polyedre("face"))
```

```
  xf=zeros(nb_s_par_face, nb_faces); yf=zeros(xf); zf=zeros(xf)
  for i=1:nb_faces
     for j=1:nb_s_par_face
        num_sommet = Polyedre("face")(i,nb_s_par_face+1-j) // pour inverser l'orientation
        v = Polyedre("coord")(:,num_sommet)
        xf(j,i)=v(1); yf(j,i)=v(2); zf(j,i)=v(3)
     end
  end
```

et c'est parti :

```
P=[ 0 0 1 1 0 0 1 1;...     // les coordonnees des sommets
    0 1 1 0 0 1 1 0;...
    0 0 0 0 1 1 1 1];

connect=[ 1 2 3 4; 5 8 7 6; 3 7 8 4;...   // les faces
          2 6 7 3; 1 5 6 2; 1 4 8 5];

Cube = tlist(["polyedre","coord","face"],P,connect);
[xf,yf,zf] = trans_polyedre(Cube);
couleurs = 2:7;   // on utilise les couleurs de la carte par defaut
xbasc(); xselect()
// le petit dessin :
plot3d(xf,yf,list(zf,couleurs),30,60,"x@y@z",[2 1 4],[-0.2,1.2,-0.2,1.2,-0.2,1.2])
```

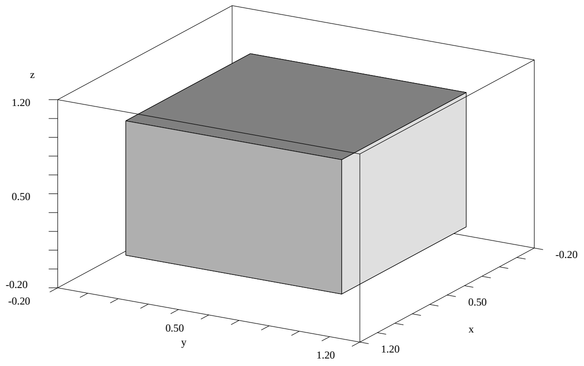


FIG. 4.7 – *Le cube...*

Pour cette utilisation `plot3d` et `plot3d1` sont équivalentes. Par contre sur un polyèdre où vous voulez des couleurs selon l'altitude (sans les calculer donc) `plot3d1` fait cela automatiquement. Dernière remarque : si vos facettes sont des triangles, il faut faire « comme si elles étaient des quadrangles ». Soient P_1,P_2,P_3 les 3 sommets d'une face, cette même face doit être décrite pour Scilab avec les 4 sommets P_1,P_2,P_3,P_1 (ce problème a été corrigé dans la version 2.5).

4.5.4 Dessiner une surface définie par $x = f_1(u,v)$, $y = f_2(u,v)$, $z = f_3(u,v)$

Réponse simple : prendre une discrétisation du domaine des paramètres et calculer les facettes ! Bien, il y a une fonction (`eval3dp`) qui fait cela automatiquement, ouf ! Cependant pour des raisons d'efficacité, la fonction qui définit le paramètrage de votre surface doit être écrite « vectoriellement ». Si $U = (u_1,u_2,\dots,u_{n_1})$ et $V = (v_1,v_2,\dots,v_{n_2})$ sont les discrétisations d'un rectangle du domaine des paramètres, votre fonction va être appelée une seule fois avec les deux « grands » vecteurs de longueur $n_1 n_2$:

```
Ug = (u1,u2,...,un1, u1,u2,...,un1, ......)
      |-----------| : cette sequence est répétée n2 fois

Vg = (v1,v1,.......,v1, v2,v2,.......,v2, ....., vn2 vn2. ... vn2)
      |--n1 fois v1--|  |--n1 fois v2--|         |--n1 fois vn2-|
```

Ce qui devrait vous permettre de comprendre la vectorisation des surfaces classiques suivantes :

```
function [x,y,z] = tore(theta, phi)
  // paramétrisation classique d'un tore de rayons R et r et d'axe Oz
  R = 1; r = 0.2
  x = (R + r*cos(phi)).*cos(theta)
  y = (R + r*cos(phi)).*sin(theta)
  z = r*sin(phi)

function [x,y,z] = helice_torique(theta, phi)
  // paramétrisation d'une helice torique
  R = 1; r = 0.3
  x = (R + r*cos(phi)).*cos(theta)
  y = (R + r*cos(phi)).*sin(theta)
  z = r*sin(phi) + 0.5*theta

function [x,y,z] = moebius(theta, rho)
  // paramétrisation d'une bande de Moëbius
  R = 1;
  x = (R + rho.*sin(theta/2)).*cos(theta)
  y = (R + rho.*sin(theta/2)).*sin(theta)
  z = rho.*cos(theta/2)

function [x,y,z] = tore_bossele(theta, phi)
  // paramétrisation d'un tore dont le petit rayon r est variable avec theta
  R = 1; r = 0.2*(1+ 0.4*sin(8*theta))
  x = (R + r.*cos(phi)).*cos(theta)
  y = (R + r.*cos(phi)).*sin(theta)
  z = r.*sin(phi)
```

et voici un exemple qui utilise la dernière surface :

```
// script pour dessiner une surface définie par des équations paramétriques
theta = linspace(0, 2*%pi, 160);
phi = linspace(0, -2*%pi, 20);
[xf, yf, zf] = eval3dp(tore_bossele, theta, phi); // calcul des facettes
xbasc()
plot3d(xf,yf,zf)
xselect()
```

Si vous voulez utiliser des couleurs et que vous ne les obtenez pas, c'est que l'orientation n'est pas la bonne : il suffit alors d'inverser le sens de l'un des deux vecteurs de la discrétisation du domaine des paramètres.

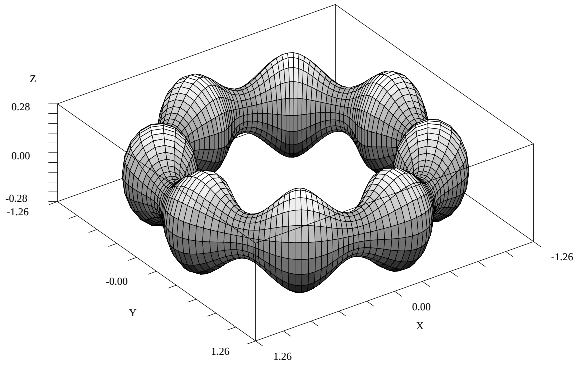


FIG. 4.8 – *Un tore bosselé...*

4.6 Les courbes dans l'espace

Pour dessiner une telle courbe l'instruction de base est `param3d`. Voici l'exemple classique de l'hélice :

```
-->t = linspace(0,4*%pi,100);

-->x = cos(t); y = sin(t) ; z = t;

-->param3d(x,y,z)  // effacer eventuellement la fenetre graphique avec xbasc()
```

mais comme cette dernière ne permet que d'affichier une seule courbe nous allons nous concentrer sur `param3d1` qui permet de faire plus de choses. Voici sa syntaxe :

```
param3d1(x,y,z,[theta,alpha,leg,flag,ebox])
param3d1(x,y,list(z,colors),[theta,alpha,leg,flag,ebox])
```

Les matrices `x, y` et `z` doivent être de même format (np,nc) et le nombre de courbes (nc) est donné par leur nombre de colonnes (comme pour `plot2d`). Les paramètres optionnels sont les mêmes que ceux de l'instruction `plot3d`. `colors` est un vecteur donnant le style pour chaque courbe (exactement comme pour `plot2d`), c'est à dire que si `colors(i)` est un entier strictement positif, la i$^{\text{ème}}$ courbe est dessinée avec la i$^{\text{ème}}$ couleur de la carte courante (ou avec différents pointillés sur un terminal noir et blanc) alors que pour une valeur entière comprise entre -9 et 0, on obtient un affichage des points (non reliés) avec le symbole correspondant. Voici un exemple qui doit vous conduire à la figure (4.9):

```
-->t = linspace(0,4*%pi,100)';

-->x1 = cos(t); y1 = sin(t) ; z1 = t;  // une helice

-->x2 = x1 + 0.2*(1-rand(x1));

-->y2 = y1 + 0.2*(1-rand(y1));

-->z2 = z1 + 0.2*(1-rand(z1));

-->xbasc() ; param3d1([x1 x2],[y1 y2],list([z1 z2],[1 -9]))

-->xset("font",4,3)   // times en gras 14 pts
```



```
-->xtitle("Helice avec perles")
```

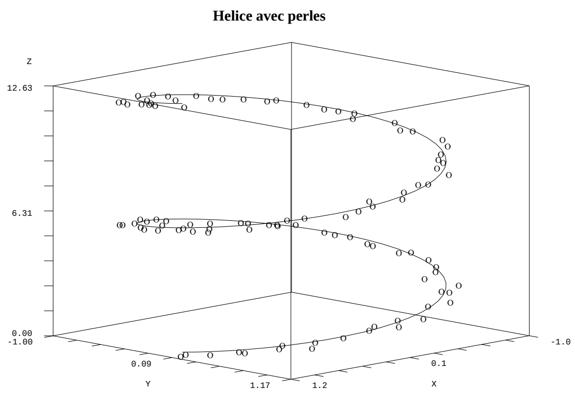


FIG. 4.9 – *Courbe et points dans l'espace...*

Comme pour `plot2d` on est obligé de l'appeler plusieurs fois si les différentes courbes à afficher n'ont pas le même nombre de points. Voici un script qui explique comment dessiner deux groupes de points avec des marques et des couleurs différentes :

```
// script pour afficher des points
n = 50;        // nombre de points
P = rand(n,3); // des points au hasard
// calcul d'une boite englobante
xmin = min(P(:,1)); xmax = max(P(:,1)) ; dx = xmax - xmin;
ymin = min(P(:,2)); ymax = max(P(:,2)) ; dy = ymax - ymin;
zmin = min(P(:,3)); zmax = max(P(:,3)) ; dz = zmax - zmin;
xvisu_min = xmin - 0.05*dx ; xvisu_max = xmax +  0.05*dx ;
yvisu_min = ymin - 0.05*dy ; yvisu_max = ymax +  0.05*dy ;
zvisu_min = zmin - 0.05*dz ; zvisu_max = zmax +  0.05*dz ;
ebox = [xvisu_min xvisu_max yvisu_min yvisu_max zvisu_min zvisu_max ] ;

// ici je separe les points en 2 groupes pour montrer comment mettre des
// symboles et des couleurs differentes pour les points
m = 40;
P1 = P(1:m,:) ; P2 = P(m+1:n,:);

// le dessin
xset("window",0)
xbasc()

// premier groupe de points
xset("pattern",2)  // du bleu avec la carte par defaut
param3d1(P1(:,1),P1(:,2),list(P1(:,3), -9),60,30,"x@y@z",[1 4],ebox)

// pour le deuxieme groupe
xset("pattern",3)  // du vert avec la carte par defaut
param3d1(P2(:,1),P2(:,2),list(P2(:,3), -5),60,30," ",[0 4])
         // -5 pour des triangles inverses
         // [0 4] ) : pour dessiner avec l'echelle courante
         //           (imposee avec le 1 er appel a param3d1)
xset("pattern",1) // pour remettre le noir comme couleur courante
[font_init] = xget("font")
```

```
xset("font",4,3)    // pour avoir le titre en time bold 14 points...
xtitle("Des points...")
xset("font",font_init(1), font_init(2)) // pour remettre la fonte initiale
xselect()
```

4.7 Dessiner plusieurs graphes dans la même fenêtre graphique

Il faut utiliser pour cela la fonction **xsetech** qui permet de sélectionner une zone rectangulaire de la fenêtre graphique courante (ce qui permet ainsi de scinder la fenêtre en plusieurs sous-fenêtres). Avant l'envoi des instructions graphiques dans une sous-fenêtre, on appelle cette fonction avec un seul argument [10] qui est un vecteur ligne à 4 composantes :

```
xsetech([xul yul dx dy])
```

qui précise la zone rectangulaire où l'on veut dessiner :

1. `xul` et `yul` permettent de définir le sommet « en haut à gauche » de la zone définissant la sous-fenêtre ;
2. `dx` et `dy` définissant la largeur et la hauteur du rectangle.

La fenêtre entière étant repérée avec le vecteur `[0 0 1 1]` (l'axe des « y » étant orienté du haut vers le bas !), un découpage horizontal en deux sous-fenêtres de même taille prendra la forme :

```
xsetech([0 0 1 0.5])  // selection de la zone superieure
instructions pour le dessin du haut
xsetech([0 0.5 1 0.5]) // selection de la zone inferieure
instructions pour le dessin du bas
```

Voici un petit script dans lequel je découpe la fenêtre graphique en 4 sous-fenêtres et qui permet d'obtenir la figure (4.10) :

```
// script pour illustrer l'emploi de xsetech
x = linspace(0,2*%pi,40);
xset("window",0)
xbasc()
xsetech([0 0 0.5 0.5])     // dessin en haut a gauche
titlepage(["    Plusieurs representations";"pour la fonction z = cos(x)sin(y)"])
xsetech([0 0.5 0.5 0.5])   // dessin en bas a gauche
xtitle("lignes isovaleurs avec contour2d")
contour2d(x,x,cos(x')*sin(x),6,1:6,"011"," ",[0 0 2*%pi 2*%pi])
xsetech([0.5 0 0.5 0.5])   // dessin en haut a droite
xtitle("vue en 3D avec plot3d1")
plot3d1(x,x,cos(x')*sin(x))
xsetech([0.5 0.5 0.5 0.5]) // dessin en bas a droite
xtitle("vue 2D en couleur avec Sgrayplot")
Sgrayplot(x,x,cos(x')*sin(x))
```

Remarque : pour les instructions `plot3d1` et `Sgrayplot`, j'utilise la carte des couleurs par défaut mais le passage en postscript (en sélectionnant noir et blanc) permet d'obtenir des niveaux de gris « continus ».

4.8 Divers

Il existe encore beaucoup de primitives graphiques dont :

1. Des variantes de plot2d, `plot2d1`, `plot2d2`,..., pour dessiner des plages constantes et ou des traits verticaux, et ou une échelle logarithmique (voir quelques exemples dans le chapitre suivant) ;

10. J'utilise ici la syntaxe simplifiée, il y a aussi des possibilités pour régler l'échelle...

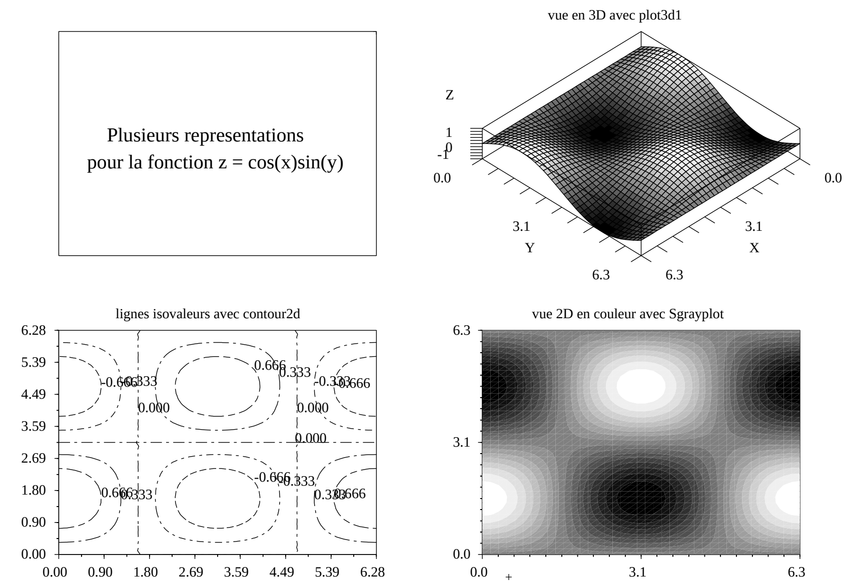


FIG. 4.10 – *Illustration pour xsetech...*

2. `contour2d` et `contour` qui permettent de dessiner des lignes isovaleurs d'une fonction $z = f(x,y)$ définie sur un rectangle ;
3. `grayplot` et `Sgrayplot` qui permettent de représenter les valeurs d'une telle fonction en utilisant des couleurs ;
4. `fec` joue le même rôle que les deux précédentes pour une fonction qui est définie sur une triangulation plane ;
5. `champ` qui permet de dessiner un champ de vecteurs en 2D.

Un dernier détail : la plupart des fonctions graphiques que j'ai évoquées dans ce chapitre admettent des variantes qui permettent de faire des graphes de fonctions plus directement si on fournit une fonction scilab comme argument. Le nom de ces fonctions commence par un f (`fplot2d,fcontour2d, fplot3d, fplot3d1, fchamp`,...). Dans le chapitre suivant nous verrons en particulier l'utilisation de `fchamp` pour dessiner le champ de vecteurs associé à une équation différentielle (pour la dimension 2).

Pour se rendre compte des possibilités il suffit de parcourir la rubrique `Graphic Library` de l'aide. Vous pouvez aussi récupérer la bibliothèque d'Enrico Ségré qui contient quelques fonctions graphiques intéressantes :

`http://www.polito.it/~segre/scistuff.html`

Chapitre 5

Quelques applications et compléments

Ce chapitre se propose de vous montrer comment résoudre certains problèmes types d'analyse numérique avec Scilab et apporte quelques compléments sur certains points (génération de nombres aléatoires). Répétons que Scilab dispose aussi de nombreuses primitives et fonctions qui permettent de résoudre pas mal de problèmes d'automatique, de traitement du signal, etc... Vous pouvez aussi récupérer différentes « boites à outils » développées par des utilisateurs de Scilab (voir la rubrique Contributions sur la Scilab Home page).

5.1 Équations différentielles

Scilab dispose d'une interface très puissante pour résoudre numériquement (de manière approchée) des équations différentielles avec la primitive `ode`. Soit donc une équation différentielle avec une condition initiale :

$$\begin{cases} u' = f(t,u) \\ u(t_0) = u_0 \end{cases}$$

où $u(t)$ est un vecteur de $\mathbb{R}^n$, f une fonction de $\mathbb{R} \times \mathbb{R}^n \longrightarrow \mathbb{R}^n$, et $u_0 \in \mathbb{R}^n$. On suppose les conditions remplies pour qu'il y ait existence et unicité de la solution jusqu'à un temps T.

5.1.1 Utilisation basique de `ode`

Dans son fonctionnement le plus basique elle est très simple à utiliser : il faut écrire le second membre f comme une fonction Scilab avec la syntaxe suivante :

```
function [f] = MonSecondMembre(t,u)
   //
   ici le code donnant les composantes de f en fonction de t et
   des composantes de u.
```

Rmq : Même si l'équation est autonome, il faut quand même mettre t comme premier argument de la fonction second membre. Par exemple voici un code possible pour le second membre de l'équation de Van der Pol :

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

et que l'on reformule comme un système de deux équations différentielles du premier ordre en posant $u_1(t) = y(t)$ et $u_2(t) = y'(t)$:

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

```
function [f] = VanDerPol(t,u)
  // second membre pour Van der Pol (c = 0.4)
  f(1) = u(2)
  f(2) = 0.4*(1 - u(1)^2)*u(2) - u(1)
```


Puis un appel a `ode` pour résoudre l'équation (l'intégrer) de t_0 à T, en partant de u_0 (un vecteur colonne), et en voulant récupérer la solution aux instants $t(1) = t_0$, $t(2)$, ..., $t(m) = T$, prendra l'allure suivante :

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

On récupère alors une « matrice » `U` de format (n,m) telle que `U(i,j)` est la solution approchée de $u_i(t(j))$ (la $i^{\text{ème}}$ composante à l'instant $t(j)$). *Rmq :* le nombre de composantes que l'on prend pour `t` (les instants pour lesquels on récupère la solution), n'a rien à voir avec la précision du calcul. Celle-ci peut se régler avec d'autres paramètres (qui ont des valeurs par défaut). D'autre part derrière `ode` il y a plusieurs algorithmes possibles, qui permettent de s'adapter à diverses situations... Pour sélectionner une méthode particulière, il faut rajouter un paramètre dans l'appel (cf le Help). Par défaut (c-a-d sans sélection explicite d'une des méthodes) on a cependant une stratégie intelligente puisque `ode` utilise initialement une méthode d'Adams prédicteur/correcteur mais est capable de changer cet algorithme par une méthode de Gear dans le cas où il détecte l'équation comme « raide[1] ».

Voici un exemple complet pour Van der Pol. Comme dans ce cas l'espace des phases est un plan, on peut déjà obtenir une idée de la dynamique en dessinant simplement le champ de vecteur dans un rectangle $[x_{min},x_{max}] \times [y_{min},y_{max}]$ avec l'instruction graphique **`fchamp`** dont la syntaxe est :

```
fchamp(MonSecondMembre,t,x,y)
```

où `MonSecondMembre` est le nom de la fonction Scilab du second membre de l'équation différentielle, `t` est l'instant pour lequel on veut dessiner le champ (dans le cas le plus courant d'une équation autonome on met une valeur sans signification, par exemple 0) et `x` et `y` sont des vecteurs lignes à nx et ny composantes, donnant les points de la grille sur lesquels seront dessinés les flèches représentant le champ de vecteur.

```
// 1/ trace du champs de vecteur issu de l'equation de Van der Pol
n = 30;
delta = 5
x = linspace(-delta,delta,n); // ici y = x
xbasc()
fchamp(VanDerPol,0,x,x)
xselect()

// 2/ resolution de l'equation differentielle
m = 500 ; T = 30 ;
t = linspace(0,T,m);  // les instants pour lesquels on recupere la solution
u0 = [-2.5 ; 2.5];    // la condition initiale
[u] = ode(u0, 0, t, VanDerPol);
plot2d(u(1,:)',u(2,:)',2,"000")
```

5.1.2 Van der Pol one more time

Dans cette partie nous allons exploiter une possibilité graphique de Scilab pour obtenir autant de trajectoires voulues sans refaire tourner le script précédent avec une autre valeur de u_0. D'autre part on va utiliser une échelle isométrique pour les dessins. Après l'affichage du champ de vecteur, chaque condition initiale sera donnée par un clic du bouton gauche de la souris[2], le pointeur étant positionné sur la condition initiale voulue. Cette possibilité graphique s'obtient avec la primitive **`xclick`** dont la syntaxe simplifiée est :

```
[c_i,c_x,c_y]=xclick();
```

1. pour faire bref on dit qu'une équation différentielle est « raide » si celle-ci s'intégre difficilement avec les méthodes (plus ou moins) explicites...
2. comme suggéré dans l'un des articles sur Scilab paru dans « Linux Magazine »

Scilab se met alors à attendre un « événement graphique » du type « clic souris », et, lorsque cet événement a lieu, on récupère la position du pointeur (dans l'échelle courante) avec `c_x` et `c_y` ainsi que le numéro du bouton avec :

valeur pour `c_i`	bouton
0	gauche
1	milieu
2	droit

Dans le script, j'utilise un clic sur le bouton droit pour sortir de la boucle des événements.

Enfin, on procède à quelques fioritures de sorte à changer de couleur pour chaque trajectoire (le tableau **couleur** me permet de sélectionner celles qui m'intéressent dans la colormap standard. Pour obtenir une échelle isométrique on utilise **fchamp** avec la liste complète des arguments :

```
fchamp(MonSecondMembre,t,x1,x2,arfact,rect,strf)
```

où **arfact** est un facteur d'échelle pour avoir la tête des flèches plus ou moins grasse (par défaut 1), **rect** et **strf** ayant la même signification que pour **plot2d** (sauf que le premier caractère est ignoré : on ne peut pas mettre de légende). L'échelle isométrique s'obtient en mettant le paramètre **y** de la chaîne **strf** à 3. Dernière fioritures : je dessine un petit rond pour bien marquer chaque condition initiale, et pour exploiter la totalité de la fenêtre graphique, j'utilise un plan de phase « rectangulaire ». Dernière remarque : lorsque le champ de vecteur apparaît vous pouvez maximiser la fenêtre graphique ! En cliquant plusieurs fois j'ai obtenu la figure (5.1) : toutes les trajectoires convergent vers une orbite périodique ce qui est bien le comportement théorique attendu pour cette équation.

```
// 1/ trace du champs de vecteur issu de l'equation de Van der Pol
n = 30;
delta_x = 6
delta_y = 4
x = linspace(-delta_x,delta_x,n);
y = linspace(-delta_y,delta_y,n);
xbasc()
fchamp(VanDerPol,0,x,y,1,[-delta_x,-delta_y,delta_x,delta_y],"031")
xselect()

// 2/ resolution de l'equation differentielle
m = 500 ; T = 30 ;
t = linspace(0,T,m);

couleurs = [21 2 3 4 5 6 19 28 32 9 13 22 18 21 12 30 27]  // 17 couleurs
num = -1
while %t
  [c_i,c_x,c_y]=xclick();
  if c_i == 0 then
    plot2d(c_x, c_y, -9, "000")  // un petit o pour marquer la C.I.
    u0 = [c_x;c_y];
    [u] = ode(u0, 0, t, VanDerPol);
    num = modulo(num+1,length(couleurs));
    plot2d(u(1,:)',u(2,:)',couleurs(num+1),"000")
  elseif c_i == 2 then
    break
  end
end
```

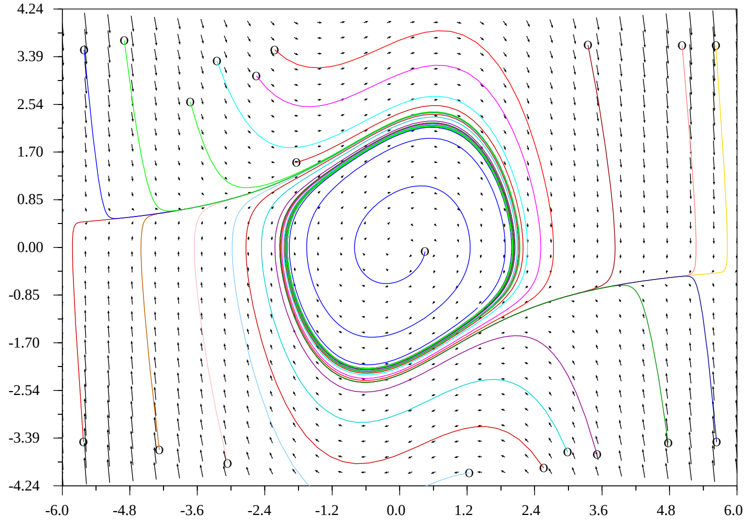



FIG. 5.1 – *Quelques trajectoires dans le plan de phase pour l'équation de Van der Pol*

5.1.3 Un peu plus d'`ode`

Dans ce deuxième exemple, nous allons utiliser la primitive **ode** avec un second membre qui admet un paramètre supplémentaire et nous allons fixer nous même les tolérances pour la gestion du pas de temps du solveur. Voici notre nouvelle équation différentielle (*Le Brusselator*) :

$$\begin{cases} \frac{du_1}{dt} = 2 - (6+\epsilon)u_1 + u_1^2 x_2 \\ \frac{du_2}{dt} = (5+\epsilon)u_1 - u_1^2 u_2 \end{cases}$$

qui admet comme seul point critique $P_{stat} = (2,(5+\epsilon)/2)$. Lorsque le paramètre ϵ passe d'une valeur strictement négative à une valeur positive, ce point stationnaire change de nature (de stable il devient instable avec pour $\epsilon = 0$ un phénomène de bifurcation de *Hopf*). On s'intéresse aux trajectoires avec des conditions initiales voisines de ce point. Voici la fonction calculant ce second membre :

```
function [f] = Brusselator(t,u,eps)
  //
  f(1) = 2 - (6+eps)*u(1) + u(1)^2*u(2)
  f(2) = (5+eps)*u(1) - u(1)^2*u(2)
```

Pour faire « passer » le paramètre supplémentaire, on remplace dans l'appel à **ode** le nom de la fonction (ici Brusselator) par une liste constituée du nom de la fonction et du ou des paramètres supplémentaires :

```
  [x] = ode(x0,t0,t,list(MonSecondMembre, par1, par2, ...))
```

Dans notre cas :

```
  [x] = ode(x0,t0,t,list(Brusselator, eps))
```

et l'on procède de même pour tracer le champ avec `fchamp`.

Pour fixer les tolérances sur l'erreur locale du solveur on rajoute les paramètres `rtol` et `atol`, juste avant le nom de la fonction second membre (ou de la liste formée par celui-ci et des paramètres supplémentaires de la fonction). À chaque pas de temps, $t_{k-1} \to t_k = t_{k-1} + \Delta t_k$, le solveur calcule une estimation de l'erreur locale e (c-a-d l'erreur sur ce pas de temps en partant de la condition initiale $v(t_{k-1}) = U(t_{k-1})$) :

$$e(t_k) \simeq U(t_k) - \left(\int_{t_{k-1}}^{t_k} f(t,v(t))dt + U(t_{k-1}) \right)$$

(le deuxième terme étant la solution exacte partant de la solution numérique $U(t_{k-1})$ obtenue au pas précédent) et compare cette erreur à la tolérance formée par les deux paramètres `rtol` et `atol` :

$$tol_i = rtol_i * |U_i(t_k)| + atol_i,\ 1 \leq i \leq n$$

dans le cas où l'on donne deux vecteurs de longueur n pour ces paramètres et :

$$tol_i = rtol * |U_i(t_k)| + atol,\ 1 \leq i \leq n$$

si on donne deux scalaires. Si $|e_i(t_k)| \leq tol_i$ pour chaque composante, le pas est accepté et le solveur calcule le nouveau pas de temps de sorte que le critère sur la future erreur ait une certaine chance de se réaliser. Dans le cas contraire, on réintègre à partir de t_{k-1} avec un nouveau pas de temps plus petit (calculé de sorte que le prochain test sur l'erreur locale soit aussi satisfait avec une forte probabilité). Comme les méthodes mise en jeu sont des méthodes « multipas[3] » le solveur, en plus du pas de temps variable, joue aussi avec l'ordre de la formule pour obtenir une bonne efficacité informatique... Par défaut les valeurs utilisées sont $rtol = 10^{-5}$ et $atol = 10^{-7}$ (sauf lorsque type selectionne une méthode de Runge Kutta). Remarque importante : le solveur peut trés bien échouer dans l'intégration...

Voici un script possible, la seule fioriture supplémentaire est un marquage du point critique avec un petit carré noir que j'obtiens avec la primitive graphique `xfrect` :

```
// Brusselator
eps = -4
P_stat = [2 ; (5+eps)/2];
// limites pour le trace du champ de vecteur
delta_x = 6; delta_y = 4;
x_min = P_stat(1) - delta_x; x_max = P_stat(1) + delta_x;
y_min = P_stat(2) - delta_y; y_max = P_stat(2) + delta_y;
n = 20;
x = linspace(x_min, x_max, n);
y = linspace(y_min, y_max, n);
// 1/ trace du champ de vecteurs
xbasc()
fchamp(list(Brusselator,eps),0,x,y,1,[x_min,y_min,x_max,y_max],"031")
xfrect(P_stat(1)-0.08,P_stat(2)+0.08,0.16,0.16) // pour marquer le point critique
xselect()

// 2/ resolution de l'equation differentielle
m = 500 ; T = 5 ;
rtol = 1.d-09; atol = 1.d-10; // tolerances pour le solveur
t = linspace(0,T,m);
couleurs = [21 2 3 4 5 6 19 28 32 9 13 22 18 21 12 30 27]
num = -1
while %t
```

3. du moins par défaut ou lorsque l'on choisit type = adams ou stiff


```
  [c_i,c_x,c_y]=xclick();
  if c_i == 0 then
    plot2d(c_x, c_y, -9, "000")  // un petit o pour marquer la C.I.
    u0 = [c_x;c_y];
    [u] = ode(u0, 0, t, rtol, atol, list(Brusselator,eps));
    num = modulo(num+1,length(couleurs));
    plot2d(u(1,:)',u(2,:)',couleurs(num+1),"000")
  elseif c_i == 2 then
    break
  end
end
```

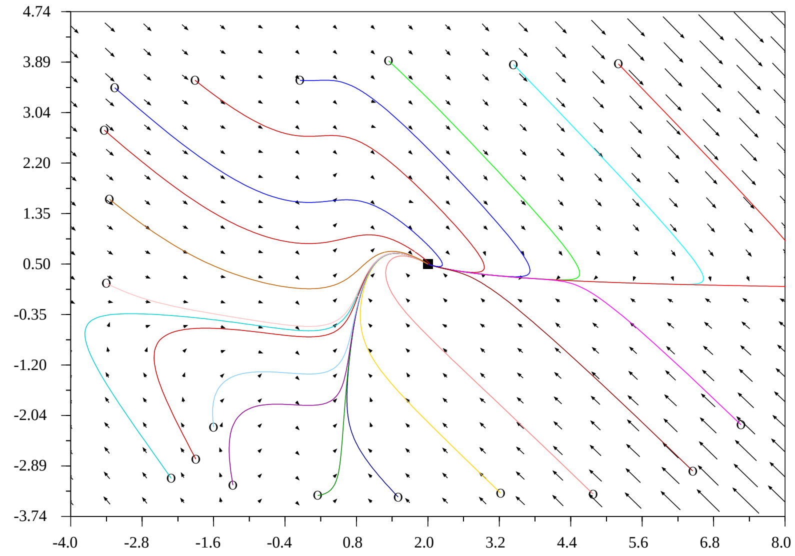


FIG. 5.2 – *Quelques trajectoires dans le plan de phase pour le Brusselator* ($\epsilon = -4$)

5.2 Génération de nombres aléatoires

5.2.1 La fonction `rand`

Jusqu'à présent elle nous a essentiellement servi à remplir nos matrices et vecteurs... Cette fonction utilise le générateur congruentiel linéaire suivant[4] :

$$X_{n+1} = f(X_n) = (aX_n + c) \bmod m,\ n \geq 0, \text{ où } \begin{cases} m = 2^{31} \\ a = 843314861 \\ c = 453816693 \end{cases}$$

4. D'après ce que j'ai cru comprendre en regardant le code source.

Sa période est bien sûr égale à m (ceci signifie que f est une permutation cyclique sur $[0,m-1]$.) Notons que tous les générateurs de nombres aléatoires sur ordinateur sont des suites parfaitement déterministes qui « apparaissent » comme aléatoires (pour les bons générateurs) selon un certain nombre de tests statistiques. Pour se ramener à des nombres réels compris dans l'intervalle $[0,1[$, on divise les entiers obtenus par m (et l'on obtient un générateur de nombres réels qui semblent suivre une loi uniforme sur $[0,1[$). Le terme initial de la suite est souvent appelé le germe et celui par défaut est $X_0 = 0$. Ainsi le premier appel à `rand` (le premier coefficient obtenu si on récupère une matrice ou un vecteur) est toujours :

$$u_1 = 453816693/2^{31} \approx 0.2113249$$

Il est cependant possible de changer le germe à tout moment avec l'instruction :

```
rand("seed",germe)
```

où germe est un entier compris dans l'intervalle (entier) $[0,m-1]$. Souvent on ressent le besoin d'initialiser la suite en choisissant un germe plus ou moins au hasard (histoire de ne pas avoir les mêmes nombres à chaque fois) et une possibilité consiste à récuperer la date et l'heure et de fabriquer le germe avec. Scilab possède une fonction `getdate` qui fournit un vecteur de 9 entiers (voir le détail avec le Help). Parmi ces 9 entiers :

- le deuxième donne le mois (1-12),
- le sizième, le jour du mois (1-31),
- le septième, l'heure du jour (0-23),
- le huitième, les minutes (0-59),
- et le neuvième, les secondes (0-61 ?).

Pour obtenir un germe je multiplie ces nombres entre eux[5] (en leur ajoutant 1 au préalable pour éviter d'obtenir le germe 0 au moins une fois sur 60...) et je multiplie le résultat obtenu par 57 de sorte que $13 \times 32 \times 24 \times 60 \times 62 \times 57 \approx 2^{31}$ ce qui donne :

```
v = getdate()
rand("seed", 57*prod(1+v([2 6 7 8 9])))
```

Noter aussi que l'on peut récupérer le germe courant avec :

```
germe = rand("seed")
```

À partir de la loi uniforme sur $[0,1[$, on peut obtenir d'autres lois et `rand` fournit aussi une interface qui permet d'obtenir la loi normale (de moyenne 0 et de variance 1). Pour passer de l'une à l'autre, on procède de la façon suivante :

```
rand("normal")  // pour obtenir la loi normale
rand("uniform") // pour revenir a la loi uniforme
```

Par défaut le générateur fournit une loi uniforme mais il est judicieux dans toute simulation de s'assurer que `rand` donne bien ce que l'on désire en utilisant l'une de ces deux instructions. On peut d'ailleurs récupérer la loi actuelle avec :

```
loi=rand("info")  // loi est l'une des deux chaînes "uniform" ou "normal"
```

Rappelons que `rand` peut s'utiliser de plusieurs façons :

1. `A = rand(n,m)` remplit la matrice `A` (n,m) de nombres aléatoires ;

5. il y a sans doute mieux à faire...

2. si `B` est une matrice déjà définie de dimensions (n,m) alors `A = rand(B)` permet d'obtenir la même chose (ce qui permet d'éviter de récupérer les dimensions de `B`) ;
3. enfin, `u = rand()` fournit un seul nombre aléatoire.

Pour les deux premières méthodes, on peut rajouter un argument supplémentaire pour imposer aussi la loi : `A = rand(n,m,loi)`, `A = rand(B,loi)`, où `loi` est l'une des deux chaînes de caractères `"normal"` ou `"uniform"`.

5.2.2 Quelques petites applications avec `rand`

À partir de la loi uniforme, il est simple d'obtenir une matrice (n,m) de nombres selon :

1. une loi uniforme sur $[a,b]$:

   ```
   X = a + (b-a)*rand(n,m)
   ```

2. une loi uniforme sur les entiers de l'intervalle $[n_1,n_2]$:

   ```
   X = floor(n1 + (n2+1-n1)*rand(n,m))
   ```

   (on tire des réels suivants une loi uniforme sur l'intervalle réel $[n_1,n_2+1[$ et on prend la partie entière).

Pour simuler une épreuve de Bernouilli avec probabilité de succès p :

```
succes = rand() < p
```

ce qui nous conduit à une méthode simple pour simuler une loi binomiale $B(N,p)$:

```
X = sum(bool2s(rand(1,N) < p))
```

(`bool2s` transforme les succès en 1 et il ne reste plus qu'à les additionner avec `sum`). Comme les itérations sont lentes en Scilab, on peut obtenir directement un vecteur (colonne) contenant m réalisations de cette loi avec :

```
X = sum(bool2s(rand(m,N) < p),"c")
```

mais on aura intérêt à utiliser la fonction `grand` qui utilise une méthode plus performante. D'autre part si vous utilisez ces petits trucs[6], il est plus clair de les coder comme des fonctions Scilab. Voici une petite fonction pour simuler la loi géométrique (nombre d'épreuves de Bernouilli nécessaires pour obtenir un succès) :

```
function [X] = G(p)
  // loi geometrique
  X = 0
  while %t
    X = X+1
    if (rand() <= p) then, break, end
  end
```

Enfin, à partir de la loi Normale $\mathcal{N}(0,1)$, on obtient la loi Normale $\mathcal{N}(\mu,\sigma^2)$ (moyenne μ et écart type σ) avec :

```
rand("normal")
X = mu + sigma*rand(n,m) // pour obtenir une matrice (n,m) de tels nombres
// ou encore en une seule instruction : X = mu + sigma*rand(n,m,"normal")
```

6. D'une manière générale on utilisera plutôt la fonction `grand` permet d'obtenir la plupart des lois classiques.

5.2.3 Dessiner une fonction de répartition empirique

Soit X une variable aléatoire continue (à valeur réelle) dont on ne connaît que m réalisations indépendantes stockées dans le vecteur $X^r = (X_1, X_2, \ldots, X_m)$. Sa *fonction de répartition* est la fonction :

$$F(x) = \text{Probabilité que } X \le x$$

et la fonction de répartition empirique définie à partir de l'échantillon X^r est définie par :

$$F_m(x) = card\{X_i <= x\}/m$$

C'est une fonction en escalier qui se calcule facilement si on trie le vecteur X^r dans l'ordre croissant (on a alors $F_m(x) = i/m$ pour $X_i \le x < X_{i+1}$). L'algorithme standard de tri de Scilab est la fonction `sort` qui trie dans l'ordre décroissant [7]. Pour trier le vecteur X^r dans l'ordre croissant, on utilise alors :

```
X_r_o = - sort(-X_r)
```

et pour éviter trop de manipulation avec `plot2d`, il en existe une variante qui dessine des plages constantes :

```
plot2d2("onn",x,y,[arguments optionnels comme pour plot2d])
```

dessinera une plage constante de valeur `y(i)` dans l'intervalle `[x(i),x(i+1)]`. Pour être sur de faire apparaître une petite plage (de valeur 0) avant X^r_{min} et une autre (de valeur 1) après X^r_{max}, on pourra alors procéder de la façon suivante :

```
function repartition_empirique(X_r)
  //
  //  trace la fonction de repartition (empirique) de
  //  X_r  un vecteur (ligne ou colonne) contenant m realisations de X
  //
  m = length(X_r)
  X_r_o = matrix(X_r,m,1)  // pour etre sur d'avoir un vecteur colonne
                           // de cette facon le code marche dans les 2 cas
  X_r_o = -sort(-X_r_o)    // X_r_o pour X_r ordonne
  // on rajoute maintenant deux points a chaque extremite :
  dx = 0.05*(X_r_o(m) - X_r_o(1))
  X_r_o = [X_r_o(1)-dx ; X_r_o ; X_r_o(m)+dx]
  // calcul des ordonnees
  y = [0 ; (1:m)'/m ; 1]
  // et le dessin
  xbasc()
  plot2d2("onn", X_r_o , y, 1, "121", "repartition empirique")
  xselect()
```

Voici maintenant un exemple qui utilise la loi normale $\mathcal{N}(0,1)$ dont la fonction de répartition peut se calculer avec la fonction erreur erf, fonction dont Scilab est muni :

$$F(x) = \int_{-\infty}^{x} \frac{1}{\sqrt{2\pi}} e^{-t^2/2} dt = \frac{1}{2}(1 + erf(\frac{x}{\sqrt{2}}))$$

```
X = rand(200,1,"normal");  // on tire 200 echantillons
repartition_empirique(X);  // dessin de la fct de repartition empirique
// les donnees pour tracer la fonction de repartition "exacte"
x = linspace(-5,5,200)';
y = (1 + erf(x/sqrt(2)))/2;
```

7. voir aussi la fonction `gsort` qui permet de faire plus de choses


```
// on rajoute la courbe sur le premier dessin
plot2d(x,y,2,"000")
```

qui doit donner quelque chose ressemblant à la figure (5.3).

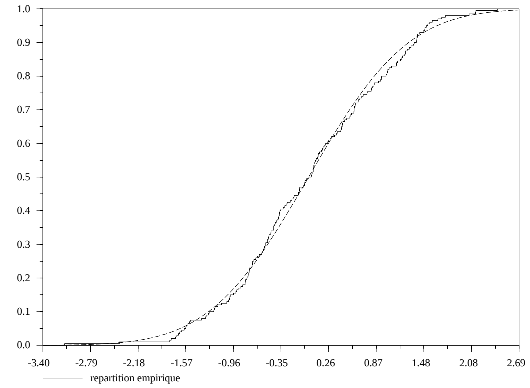


FIG. 5.3 – *Fonctions de répartition exacte et empirique pour la loi normale*

5.2.4 Dessiner un histogramme

La fonction scilab adéquate s'appelle `histplot` et sa syntaxe est la suivante :

```
histplot(n, X,[arguments optionnels comme pour plot2d])
```

où `n` est soit un entier, soit un vecteur ligne (avec $n_i < n_{i+1}$) et `X` le vecteur (ligne ou colonne) des données à examiner :

1. dans le cas où `n` est un vecteur ligne, les données sont comptabilisées selon les k classes $C_i = [n_i, n_{i+1}[$ (le vecteur `n` à donc $k+1$ composantes) : la plage de l'histogramme correspondant à cet intervalle vaudra alors (m étant le nombre de données) :

$$\frac{\text{card } \{X_j \in C_i\}}{m}$$

2. dans le cas où `n` est un entier, les données sont comptabilisées dans les n classes équidistantes :

$$C_1 = [c_1, c_2],\ C_i =]c_i, c_{i+1}],\ i = 2,...,n, \text{ avec } \left\{ \begin{array}{l} c_1 = min(X),\ c_{n+1} = max(X) \\ \Delta c = (c_{n+1} - c_1)/n \end{array} \right.$$

Voici un petit exemple, toujours avec la loi normale (cf figure (5.4)) :

```
X = rand(100000,1,"normal"); classes = linspace(-5,5,21);
histplot(classes,X)
// on lui superpose le tracé de la densité de N(0,1)
x = linspace(-5,5,60)'; y = exp(-x.^2/2)/sqrt(2*%pi);
plot2d(x,y,2,"000")
```

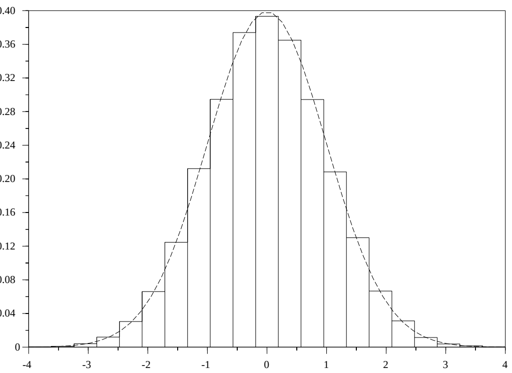


FIG. 5.4 – *Histogramme d'un échantillon de nombres aléatoires suivants* $N(0{,}1)$

5.2.5 La fonction `grand`

Pour des simulations lourdes qui utilisent beaucoup de nombres aléatoires la fonction standard `rand` avec sa période de $2^{31} (\simeq 2.147\ 10^9)$ est peut être un peu juste. Depuis la version 2.4 Scilab est équipé d'un générateur de luxe `grand` qui permet aussi de simuler toutes les lois classiques. `grand` s'utilise presque de la même manière que `rand`, c-a-d que l'on peut utiliser l'une des deux syntaxes suivantes (pour la deuxième il faut évidemment que la matrice `A` soit définie au moment de l'appel) :

```
grand(n,m,loi, [p1, p2, ...])
grand(A,loi, [p1, p2, ...])
```

où `loi` est une chaîne de caractères précisant la loi, celle-ci étant suivie de ses paramètres éventuels. Quelques exemples (pour obtenir un échantillon de m réalisations, sous la forme d'un vecteur colonne) :

1. une loi uniforme sur les entiers de l'intervalle $[1{,}2147483562]$:
   ```
   X = grand(m,1,"lgi")
   ```
   (ce type de générateur constitue la base de `grand`, voir plus loin) ;
2. une loi uniforme sur les entiers de l'intervalle $[n1{,}n2]$:
   ```
   X = grand(m,1,"uin",n1,n2)
   ```
   (il faut que $n2 - n1 \leq 2147483561$ mais dans le cas contraire ce problème est signalé par un message d'erreur) ;
3. pour la loi uniforme sur $]0{,}1[$ (c-a-d que 0 et 1 ne sont jamais obtenus) :
   ```
   X = grand(m,1,"def")
   ```
   (celui-ci est obtenu en divisant les nombres issus du générateur basique par 2147483563)
4. pour la loi uniforme sur $]a{,}b[$ (a et b ne sont jamais obtenus) :
   ```
   X = grand(m,1,"unf",a,b)
   ```
5. pour la loi binomiale $B(N{,}p)$:
   ```
   X = grand(m,1,"bin",N,p)
   ```
6. pour la loi de Poisson de moyenne μ :
   ```
   X = grand(m,1,"poi",mu)
   ```
7. pour la loi normale de moyenne μ et d'écart type σ :
   ```
   X = grand(m,1,"nor",mu,sigma)
   ```


Il y en a bien d'autres (cf la page d'aide).

La base de `grand` est constituée par un générateur avec une très longue période ($2.3\ 10^{18}$), ce générateur étant scindé en 32 générateurs « virtuels », chacun d'eux pouvant fournir 2^{20} =1 048 576 « blocs » de nombres, chaque bloc contenant 2^{30} =1 073 741 824 nombres appartenant à l'intervalle entier [1 , 2 147 483 562] (d'après la page d'aide...!!). On peut passer d'un générateur virtuel à l'autre avec :

```
grand('setcgn',G)
```

où $G = 1{,}2,\dots,32$ et récupérer le générateur (virtuel) courant avec :

```
G = grand('getcgn',G)
```

(par défaut $G = 1$) et il y a aussi des possibilités d'initialisation (il faut 2 germes). Bref il y a de quoi faire...

5.2.6 Les fonctions de distributions classiques et leurs inverses

Ces fonctions sont souvent utiles pour les tests statistiques (χ^2_r, ...) car elles permettent de calculer, soit :

1. la fonction de répartition en 1 ou plusieurs points ;
2. son inverse en 1 ou plusieurs points ;
3. l'un des paramètres de la loi, étant donnés les autres et un couple $(x{,}F(x))$;

Dans le Help, vous les trouverez à la rubrique « Cumulative Distribution Functions... », toutes ces fonctions commencent par les lettres `cdf`. Prenons par exemple la loi normale $\mathcal{N}(\mu{,}\sigma^2)$, la fonction qui nous intéresse s'appelle `cdfnor` et la syntaxe est alors :

1. `[P,Q]=cdfnor("PQ",X,mu,sigma)` pour obtenir $P = F_{\mu,\sigma}(X)$ et $Q = 1-P$, `X`, `mu` et `sigma` peuvent être des vecteurs (de même taille) et l'on obtient alors pour `P` et `Q` des vecteurs avec $P_i = F_{\mu_i,\sigma_i}(X_i)$;
2. `[X]=cdfnor("X",mu,sigma,P,Q)` pour obtenir $X = F^{-1}_{\mu,\sigma}(P)$ (de même que précédemment les arguments peuvent être des vecteurs de même taille et l'on obtient alors $X_i = F^{-1}_{\mu_i,\sigma_i}(P_i)$;
3. `[mu]=cdfnor("Mean",sigma,P,Q,X)` pour obtenir la moyenne ;
4. et finalement `[sigma]=cdfnor("Std",P,Q,X,mu)` pour obtenir l'écart type.

Ces deux dernières syntaxes fonctionnant aussi si les arguments sont des vecteurs de même taille. Ainsi dans mon exemple sur le dessin d'une fonction de répartition (où j'ai exprimé la fonction de répartition de la loi normale $\mathcal{N}(0{,}1)$ avec la fonction erreur), j'aurais pu utiliser plus directement :

```
[y,z] = cdfnor("PQ",x,0*ones(x),1*ones(x));
```

5.2.7 Un test du χ^2

On va réaliser un tel test sur un exemple d'école qui consiste à effectuer N tirages dans l'urne de *Polya*. Celle-ci contient au départ r boules rouges et v boules vertes et chaque tirage consiste à tirer une boule au hasard et à la remettre dans l'urne avec c boules de la même couleur. On note X_k la proportion de boules vertes après k tirages et V_k le nombre de boules vertes :

$$X_0 = \frac{v}{v+r}, V_0 = v.$$

Si l'on choisit $v = r = c = 1$, on a les résultats suivants que l'on va essayer de retrouver par simulation (pour les deux premiers) :

1. $E(X_N) = E(X_0) = X_0 = 1/2$;
2. X_N suit une loi uniforme sur $\{\frac{1}{N+2},\dots,\frac{N+1}{N+2}\}$;
3. pour $N \to +\infty$, X_N converge p.s. vers la loi uniforme sur $[0{,}1)$.

Fonction Scilab de base

Pour effectuer différentes simulations, on peut programmer une fonction prenant le paramètre N et qui effectue N tirages successifs. Cette fonction renvoie alors X_N et V_N :

```
function [XN, VN] = Urne_de_Polya(N)
  // simulation de N tirages d'une "Urne de Polya" :
  //
  VN = 1 ; V_plus_R = 2 ; XN = 0.5
  for i=1:N
    u = rand()   // tirage d'une boule
    V_plus_R = V_plus_R + 1   // ca fait une boule de plus
    if (u <= XN) then  // on a tire une boule verte : on a XN proba de
                       // tomber sur une verte (et 1 - XN sur une rouge)
      VN = VN + 1
    end
    XN = VN / V_plus_R // on actualise la proportion de boules vertes
  end
```

Pour effectuer des statistiques conséquentes cette fonction va être appelée trés souvent et comme les itérations sont lentes en Scilab (comme dans tous ces langages...), on peut écrire une fonction capable de simuler m processus en « parallèle » (il ne s'agit pas de vrai parallélisme informatique mais simplement d'exploiter le fait que les opérations matricielles sont efficaces en Scilab). Voici la fonction correspondante (dans laquelle la fonction `find` récupère les indices « vrai » d'un vecteur de booléens) :

```
function [XN, VN] = Urne_de_Polya_parallele(N,m)
  // simulation de m processus en // de N tirages d'une << Urne de Polya >> :
  //
  VN = ones(m,1) ; V_plus_R = 2 ; XN = 0.5*ones(m,1)
  for i=1:N
    u = rand(m,1)   // tirage d'une boule (ds chacune des m urnes)
    V_plus_R = V_plus_R + 1  // ca fait une boule de plus (qq soit l'urne)
    ind = find(u <= XN)      // trouve les numeros des urnes pour lesquels
                             // on a tire une verte
    VN(ind) = VN(ind) + 1    // on augmente alors le nb de boules vertes de ces urnes
    XN = VN / V_plus_R       // on actualise la proportion de boules vertes
  end
```

Le script Scilab

Dans ce script, on se propose de retrouver par simulation le résultat attendu pour l'espérance puis de tester l'hypothèse H sur le comportement de la variable aléatoire X_N « H : X_N suit une loi uniforme sur $\{\frac{1}{N+2},\ldots\ldots,\frac{N+1}{N+2}\}$ ». Pour cela on effectue m simulations puis :

1. on affiche alors l'espérance empirique obtenue (en l'accompagnant d'un intervalle empirique d'erreur à 95% obtenu via le calcul de la variance empirique) ;
2. pour tester l'hypothèse H, on effectue un test du χ_2 : on compte le nombre d'occurences `occ(i)` pour chacun des $N+1$ résultats possibles et l'on calcule alors la quantité :

$$Y = \frac{\sum_{i=1}^{N+1}(occ_i - mp_i)^2}{mp_i}$$

qui tend à être grande si l'hypothèse testée n'est pas la bonne (p_i est la probabilité théorique d'obtenir le résultat i, mais ici on s'attend à une loi uniforme pour chacun des $N+1$ résultats possibles et donc $p_i = p = 1/(N+1)$). On montre que lorsque $m \to +\infty$, Y suit une loi du χ^2 à N

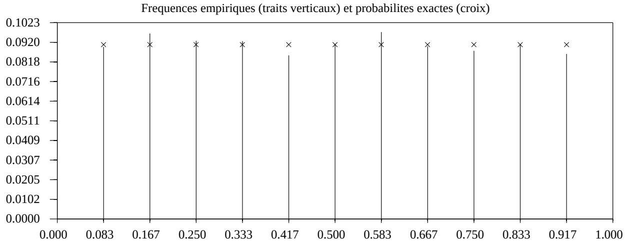
degré de liberté. Si on suppose que le nombre de simulations m est suffisamment grand pour que l'on soit assez proche de la loi asymptotique attendue, on rejette l'hypothèse H si $Y > F^{-1}(1-\alpha)$ avec $\alpha = 0{,}05$ par exemple (où F est la fonction de répartition du χ^2 à N degrés de liberté).

```
// polya simulation :
N = 10;
m = 5000;
[XN, VN] = Urne_de_Polya_parallele(N,m);
EN = sum(XN)/m;                          // esperance empirique
sigma = sqrt(sum((XN - EN).^2)/(m-1));  // variance empirique
delta =  2*sigma/sqrt(m);               // increment pour l'intervalle empirique
// presentation du resultat pour l' esperance
write(%io(2)," E exact = "+string(0.5))
write(%io(2)," E estime = "+string(EN))
write(%io(2)," Intervalle de confiance a 95% : ["+string(EN-delta)+","+string(EN+delta)+"]")

// mise en place du test
alpha = 0.05;
p = 1/(N+1);  //  proba theorique pour chaque resultat (loi uniforme)

// 1/ calcul du nb d'occurences de chacun des N+1 resultats possibles :
occ = zeros(N+1,1);
for i=1:N+1
  occ(i) = sum(bool2s(XN == i/(N+2)));
end
// petite verif :
if sum(occ) ~= m then, error(" Probleme..."), end

// 2/ calcul de la quantite test Y
Y = sum( (occ - m*p).^2 / (m*p) );

// 3/ les grandes valeurs de Y doivent etre rejetees
//    c-a-d celles pour lesquelles F_repart_du_chi2_a_N_ddl(Y) > 1 - seuil
//    La valeur du seuil (en Y) se calcule par l'inverse de la fonction de repartition :
Y_seuil = cdfchi("X",N,1-alpha,alpha);

// 4/ affichage des resultats
//
write(%io(2)," Test du chi 2 : ")
write(%io(2)," -------------- ")
write(%io(2)," valeur obtenue par le test : "+string(Y))
write(%io(2)," valeur seuil a ne pas depasser : "+string(Y_seuil))
if (Y > Y_seuil) then
  write(%io(2)," Conclusion provisoire : Hypothese rejetee !")
else
  write(%io(2)," Conclusion provisoire : Hypothese non rejetee !")
end

// 5/ dessins...
deff("[d] = d_chi2(X,N)","d = X.^(N/2 - 1).*exp(-X/2)/(2^(N/2)*gamma(N/2))");
xbasc()
frequences = occ/m;
ymax = max([frequences ; p]);
rect = [0 0 1 ymax*1.05];
```

```
xsetech([0,0,1,0.5])
plot2d3("onn",(1:N+1)'/(N+2), frequences, 2, "011", " ", rect, [0 N+2 0 10])
plot2d((1:N+1)'/(N+2),p*ones(N+1,1),-2,"000")
xtitle("Frequences empiriques (traits verticaux) et probabilites exactes (croix)")
xselect()
xsetech([0.33,0.5,0.33,0.5])
// on trace la densite chi2 a N ddl
X = linspace(0,1.1*max([Y_seuil Y]),50);
D = d_chi2(X,N);
plot2d(X,P,1,"122","densite chi2")
plot2d3("gnn",[Y Y_seuil],[d_chi2(Y,N) d_chi2(Y_seuil,N)],[2 3],"000")
xstring(Y_seuil,1.2*d_chi2(Y_seuil,N),"seuil")
xtitle("Positionnement de Y  par rapport a la valeur seuil")
xselect()
```

Voici un résultat obtenu avec $N = 10$ et $m = 5000$ (cf figure (5.5)) :

```
E exact = 0.5
E estime = 0.4959167
Intervalle de confiance a 95% : [0.4884901,0.5033433]

Test du chi 2 :
--------------
valeur obtenue par le test : 8.3176
valeur seuil a ne pas depasser : 18.307038
Conclusion provisoire : Hypothese non rejetee !
```



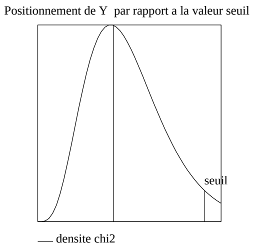


FIG. 5.5 – *Illustration pour le test du χ^2 de l'urne de Polya...*

5.2.8 Test de Kolmogorov-Smirnov

Ce test est plus naturel que celui du χ^2 lorsque la loi attendue a une fonction de répartition continue. Soit X une v.a. réelle dont la loi a une fonction de répartition continue F et X_1, X_2,..., X_m, m réalisations indépendantes d'un processus dont on suppose qu'il suit la même loi que X (on cherche à tester cette hypothèse). Le test consiste à mesurer l'écart entre la fonction de répartition exacte et la fonction de répartition empirique :

$$K_m = \sqrt{m} \sup_{-\infty<x<+\infty} |F(x) - F_m(x)|$$

et à le comparer à une valeur « admissible ». À la limite, la loi de la variable aléatoire K_m a la fonction de répartition suivante :

$$\lim_{m\to+\infty} P(K_m \le x) = H(x) = 1 - 2\sum_{j=1}^{+\infty}(1)^{j-1}e^{-2j^2x^2}$$

et de même que pour le test du χ^2 on rejette l'hypothèse lorsque :

$$K_m > H^{-1}(1-\alpha)$$

avec $\alpha = 0.05$ par exemple. Si on utilise l'approximation $H(x) \simeq 1 - e^{-2x^2}$. alors la valeur seuil à ne pas dépasser est :

$$K_{seuil} = \sqrt{\frac{1}{2}\ln(\frac{1}{\alpha})}$$

On dispose cependant d'une expression asymptotique pour ce seuil correspondant à la loi exacte suivie par K_m (suffisamment précise pour $m > 30$) :

$$K_{seuil} = \sqrt{\frac{1}{2}\ln(\frac{1}{\alpha})} - \frac{1}{6\sqrt{m}} + O(\frac{1}{m})$$

Le calcul de K_m ne pose pas de problème si on trie le vecteur $X^r = (X_1,\ X_2,\ \ldots,\ X_m)$. Supposons ce tri effectué, en remarquant que :

$$\sup_{x\in[X_i,X_{i+1}[} F_m(x) - F(x) = \frac{i}{m} - F(X_i), \text{ et } \sup_{x\in[X_i,X_{i+1}[} F(x) - F_m(x) = F(X_{i+1}) - \frac{i}{m}$$

les deux quantités suivantes se calculent facilement :

$$K_m^+ = \sqrt{m} \sup_{-\infty<x<+\infty} (F_m(x) - F(x)) = \sqrt{m} \max_{1\le j\le m} (\frac{j}{m} - F(X_j))$$

$$K_m^- = \sqrt{m} \sup_{-\infty<x<+\infty} (F(x) - F_m(x)) = \sqrt{m} \max_{1\le j\le m} (F(X_j) - \frac{j-1}{m})$$

et l'on obtient alors $K_m = \max(K_m^+, K_m^-)$.

Nous allons illustrer ce test sur le processus stochatisque suivant (où U désigne la loi uniforme sur [0,1]) :

$$X_n(t) = \frac{1}{\sqrt{n}}\sum_{i=1}^{n}(1_{\{U_i\le t\}} - t)$$

qui est tel que pour $t \in]0,1[$ fixé, on a :

$$\lim_{n\to+\infty} X_n(t) = \mathcal{N}(0,t(1-t))$$

Pour cela on se propose de prendre n « assez grand » et de réaliser m simulations à n fixé et finalement d'observer la convergence en loi, d'abord graphiquement (en superposant la fonction de répartition empirique avec la fonction de répartition exacte) puis en réalisant ce test de *Kolmogorov-Smirnov*. Avec les raccourcis d'écriture permis en Scilab, la fonction principale pour obtenir une réalisation de $X_n(t)$ peut s'écrire de la façon suivante :

```
function [X] = pont_brownien(t,n)
  //
  X = sum(bool2s(grand(n,1,"def") <= t) - t)/sqrt(n)
```

Voici maintenant un script possible, qui, après m simulations du processus, affiche le graphe de la fonction de répartition empirique puis lui superpose celui de la fonction de répartition attendue et finalement effectue le test statistique :

```
// Le pont Brownien
// Petite simulation pour illustrer que : Xn(t) --> N(0,t(1-t))  pour n -> infini
//
// où  Xn(t) = somme_{i=1}^n ( 1_(Ui <= t) - t ) /sqrt(n)
//
t = 0.3;
sigma = sqrt(t*(1-t));  // l'écart type attendu
n = 1000;               // n "grand"
m = 4000;               // le nb de simulations
X = zeros(m,1);         // initialisation du vecteur des realisations
for k=1:m
  X(k) = pont_brownien(t,n);  // la boucle pour calculer les realisations
end
repartition_empirique(X)      // le dessin de la fonction de repartition empirique
x = linspace(min(X),max(X),60)';               // les abscisses et
[P,Q]=cdfnor("PQ",x,0*ones(x),sigma*ones(x));  // les ordonnees pour la fonction exacte
plot2d(x,P,2,"000")                            // on l'ajoute sur le dessin initial

// mise en place du test KS
alpha = 0.05
X = - sort(-X); // tri
FX = cdfnor("PQ",X,0*ones(X),sigma*ones(X));
Dplus = max( (1:m)'/m - FX );
Dmoins = max( FX - (0:m-1)'/m );
Km = sqrt(m)*max([Dplus ; Dmoins]);
K_seuil = sqrt(log(1/alpha)/2) - 1/(6*sqrt(m)) ;

// affichage des resultats
//
write(%io(2)," Test KS : ")
write(%io(2)," -------- ")
write(%io(2)," valeur obtenue par le test : "+string(Km))
write(%io(2)," valeur seuil a ne pas depasser : "+string(K_seuil))
if (Km > K_seuil) then
  write(%io(2)," Conclusion provisoire : Hypothese rejetee !")
else
  write(%io(2)," Conclusion provisoire : Hypothese non rejetee !")
end
```

Voici les résultats obtenus avec $n = 1000$ et $m = 4000$ (cf figure(5.6)) :

```
Test KS :
--------
valeur obtenue par le test : 1.1204036
valeur seuil a ne pas depasser : 1.2212382
Conclusion provisoire : Hypothese non rejetee !
```

Pour finir nous allons, pour n fixé, dessiner la courbe $X_n(t), t \in [0,1]$. Il est assez simple de voir, qu'après le tri croissant des U_i (en supposant aussi les U_i tous distincts et différents de 0 et 1 et en

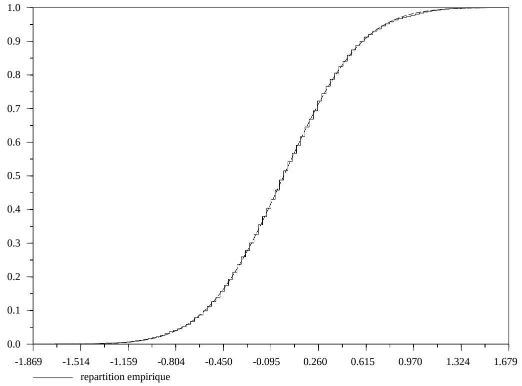


FIG. 5.6 – *fonctions de répartition empirique et exacte*

posant $U_0 = 0$ et $U_{n+1} = 1$) que :

$$X_n(t) = \frac{i - nt}{\sqrt{n}}, \text{ pour } U_i \leq t < U_{i+1}$$

On peut alors calculer cette courbe (en reliant les discontinuités par des segments verticaux) avec la fonction suivante :

```
function [] = dessin_pont_brownien(n)
  //
  U = -sort(-rand(1,n))
  Xbas = ((0:n-1) - n*U)/sqrt(n)
  Xhaut = ((1:n) - n*U)/sqrt(n)
  absc = [0 ; matrix([U ; U], 2*n, 1) ; 1]
  ord  = [0 ; matrix([Xbas ; Xhaut], 2*n, 1) ; 0]
  xbasc()
  plot2d(absc,ord)
```

La figure (5.7) montre une courbe obtenue avec $n = 10000$ avec 3 zooms successifs qui peuvent laisser entrevoir le caractère fractal de la « courbe limite ».

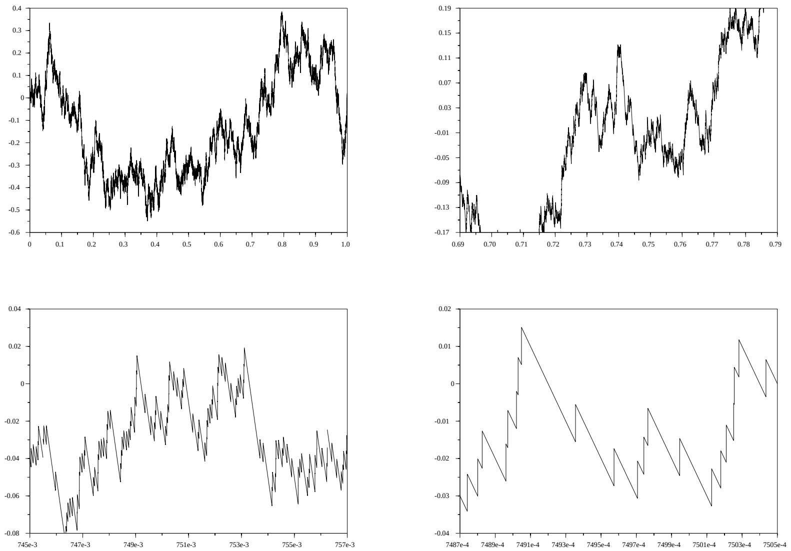


FIG. 5.7 – *Un pont Brownien...*

Chapitre 6

Bétisier

Cette partie essaie de répertorier quelques erreurs fréquentes que l'on peut commettre en Scilab...

6.1 Définition d'un vecteur ou d'une matrice « coefficient par coefficient »

Cette erreur est l'une des plus fréquentes. Considérons le script suivant :

```
K = 100  // le seul parametre de mon script
for k=1:K
   x(i) = quelque chose
   y(i) = autre chose
end
plot(x,y)
```

Lorsque l'on exécute ce script pour la première fois, on définit les deux vecteurs `x` et `y` de façon assez naturelle et tout semble fonctionner... Il y a déjà un petit défaut car, à chaque itération, Scilab redéfinit les dimensions de ces vecteurs (il ne sait pas que leur taille finale sera (K,1)). Notons aussi que, par défaut, il va créer des vecteurs colonnes. Lors de la deuxième exécution (je viens de changer le paramètre `K`...) les vecteurs `x` et `y` sont connus et tant que `k` est inférieur à 100 (la valeur initiale de `K`) il se contente de changer la valeur des composantes. Par conséquent si la nouvelle valeur de `K` est telle que :

- `K <100` alors nos vecteurs `y` et `y` ont toujours 100 composantes (seules les `K` premières ont été modifiées) et le dessin ne représentera pas ce que l'on veut ;
- `K >100` on a apparemment pas de problèmes (mis à part le fait que la taille de vecteurs est de nouveau à chaque fois différente à partir de l'itération 101).

La bonne méthode est de définir complètement les vecteurs `x` et `y` avec une initialisation du genre :

```
x = zeros(K,1) ; y = zeros(K,1)
```

et l'on ne retrouve plus ces défauts. Notre script s'écrira donc :

```
K = 100  // le seul parametre de mon script
x = zeros(K,1); y = zeros(K,1);
for k=1:K
   x(i) = quelque chose
   y(i) = autre chose
end
plot(x,y)
```

6.2 Apropos des valeurs renvoyées par une fonction

Supposons que l'on ait programmé une fonction Scilab qui renvoie deux arguments, par exemple :

```
function [x1,x2] = resol(a,b,c)
```

```
// resolution de l'equation du second degre a x^2 + b x + c = 0
// formules ameliorees pour plus de robutesse numerique
// (en evitant la soustraction de 2 nombres voisins)
if (a == 0) then
  error(" on ne traite pas le cas a=0 !")
else
  delta = b^2 - 4*a*c
  if (delta < 0) then
    error(" on ne traite pas le cas ou delta < 0 ")
  else
    if (b < 0) then
      x1 = (-b + sqrt(delta))/(2*a) ; x2 = c/(a*x1)
    else
      x2 = (-b - sqrt(delta))/(2*a) ; x1 = c/(a*x2)
    end
  end
end
```

D'une manière générale, lorsque l'on invoque une fonction à partir de la fenêtre Scilab de la façon suivante :

```
-->resol(1.e-08, 0.8, 1.e-08)
 ans  =

  - 1.250D-08
```

celui-ci est mis dans la variable `ans`. Mais `ans` est toute seule et comme cette fonction renvoie 2 valeurs, seule la première est affectée dans `ans`. Pour récupérer les deux valeurs, on utilise la syntaxe :

```
-->[x1,x2] = resol(1.e-08, 0.8, 1.e-08)
 x2  =

  - 80000000.
 x1  =

  - 1.250D-08
```

Un autre piège, plus pervers, est le suivant. Supposons que l'on fasse une petite étude sur la précision obtenue avec ces formules (vis à vis de celle obtenue avec les formules classiques). Pour un jeu de valeurs pour a et c (par exemple $a = c = 10^{-k}$ en prenant plusieurs valeurs pour k), on va calculer toutes les racines obtenues et les mettre dans deux vecteurs à des fins ultérieures d'analyse. Il semble naturel de procéder de cette façon :

```
b = 0.8;
kmax = 20;
k = 1:kmax;
x1 = zeros(kmax,1); x2=zeros(kmax,1);
for i = 1:kmax
  a = 10^(-k(i)); // c = a
  [x1(i), x2(i)] = resol(a, b, a); // ERREUR !
end
```

Mais ceci ne marche pas (alors que si la fonction renvoie une seule valeur c'est OK). Il faut procéder en deux temps :

```
  [rac1, rac2] = resol(a, b, a);
  x1(i) = rac1; x2(i) = rac2;
```


6.3 Je viens de modifier ma fonction mais...

tout semble se passer comme avant la modification ! Vous avez peut être oublié de sauvegarder les modifications avec votre éditeur ou, plus certainement, vous avez oublié de recharger le fichier qui contient cette fonction dans Scilab avec l'instruction `getf` ! Une petite astuce : votre instruction `getf` n'est certainement pas très loin dans l'historique des commandes, taper alors sur la touche ↑ jusqu'à la retrouver.

6.4 Problème avec `rand`

Par défaut `rand` fournit des nombres aléatoires selon la loi uniforme sur [0,1[mais on peut obtenir la loi normale $\mathcal{N}(0,1)$ avec : `rand("normal")`. Si on veut de nouveau la loi uniforme il ne faut pas oublier l'instruction `rand("uniform")`. Un moyen imparable pour éviter ce problème est de préciser la loi à chaque appel (voir chapitre précédent).

6.5 Vecteurs lignes, vecteurs colonnes...

Dans un contexte matriciel ils ont une signification précise mais pour d'autres applications il semble naturel de ne pas faire de différence et d'adapter une fonction pour qu'elle marche dans les deux cas. Cependant pour effectuer les calculs rapidement, il est préférable de recourir à des expressions matricielles plutôt qu'à des itérations et il faut alors choisir une forme ou l'autre. On peut utiliser alors la fonction `matrix` de la façon suivante :

```
x = matrix(x,1,length(x)) // pour obtenir un vecteur ligne
x = matrix(x,length(x),1) // pour obtenir un vecteur colonne
```

Si on cherche simplement à obtenir un vecteur colonne, on peut utiliser le raccourci :

```
x = x(:)
```

6.6 Opérateur de comparaison

Dans certains cas Scilab admet le symbole `=` comme opérateur de comparaison :

```
-->2 = 1
 Warning: obsolete use of = instead of ==
    !
 ans  =

  F
```

mais il vaut mieux toujours utiliser le symbole `==`.

6.7 Primitives et fonctions Scilab

Dans ce document j'ai utilisé plus ou moins indifféremment les termes de primitive et fonction pour désigner des « procédures » offertes par la version courante de Scilab. Il existe cependant une différence fondamentale entre une primitive qui est codée en fortran 77 ou en C et une fonction (appelée aussi macro) qui est codée en langage Scilab : une fonction est considérée comme une variable Scilab, et, à ce titre vous pouvez faire passer une fonction en tant qu'argument d'une autre fonction. Il n'en est pas de même pour une primitive. Par exemple, la plupart des fonctions mathématiques courantes (exp, cos, sin,...) sont des primitives et non des fonctions Scilab. Il faut bien tenir compte de ce fait lorsque l'on utilise une fonction qui attend un tel argument. Par exemple, voici une petite fonction pour calculer une intégrale par la méthode de *Monte Carlo* :

```
function [I,sigma]=MonteCarlo(a,b,f,n)
  //           /b
  //  approx de |  f(x) dx par la methode de Monte Carlo
  //           /a
  //  on renvoie aussi l'ecart type empirique
  //  f doit etre codee comme une fonction scilab et de sorte
  //  a admettre un argument de type vecteur.
  //  n est le nombre de nombres aleatoires utilises
  u = (b-a)*rand(n,1,"uniform") + a
  y = f(u)
  Moyenne = sum(y)/n
  I = (b-a)*Moyenne
  sigma = (b-a) * sqrt(sum( (y - Moyenne).^2 )/(n-1))
```

Supposons que l'on veuille tester cette fonction en intégrant l'exponentielle entre 0 et 1 ; on ne peut pas utiliser la méthode suivante :

```
-->[I,sigma]=MonteCarlo(0,1,exp,10000)
                           !--error    25
bad call to primitive :exp
```

Il faut alors coder l'exponentielle comme une fonction Scilab pour obtenir l'effet cherché comme dans le script suivant :

```
// integration par Monte Carlo
n = 10000 ;
a = 0 ;
b = 1 ;
deff("[y]=f1(x)","y = exp(x)")  // f1 est l'exponentielle codee en scilab
I_exact = %e - 1;
[I, sigma] = MonteCarlo(a,b,f1,n);
delta = 2*sigma/sqrt(n);
// presentation du resultat
write(%io(2)," I exact = "+string(I_exact))
write(%io(2)," I approche = "+string(I))
write(%io(2),...
" Intervalle de confiance empirique a 95% : ["+string(I-delta)+","+string(I+delta)+"]")
```

qui peut donner le résultat suivant :

```
 I exact = 1.7182818
 I approche = 1.7182109
 Intervalle de confiance empirique a 95% : [1.70843,1.7279917]
```

6.8 Évaluation d'expressions booléennes

Contrairement au langage C, l'évaluation des expressions booléennes de la forme :

$$a \text{ ou } b$$
$$a \text{ et } b$$

passe d'abord par l'évaluation des sous-expressions booléennes a et b avant de procéder au « ou » pour la première ou au « et » pour la deuxième (dans le cas où a renvoie « vrai » pour la première (et faux

pour la deuxième) on peut se passer d'évaluer l'expression boolénne b) (cf Priorité des opérateurs dans le chapitre « Programmation »).

6.9 Nombres Complexes et nombres réels

Tout est fait dans Scilab pour traiter de la manière les réels et les complexes ! Ceci est assez pratique mais peut conduire à certaines surprises... (à compléter!)

That 's all Folks...

Annexe A

Correction des exercices du chapitre 2

1.

```
--> n  = 5 // pour  fixer une valeur a n...
--> A = 2*eye(n,n) - diag(ones(n-1,1),1) - diag(ones(n-1,1),-1)
```

Un moyen plus rapide consiste à utiliser la fonction `toeplitz` :

```
-->n=5; // pour  fixer une valeur a n...

-->toeplitz([2 -1 zeros(1,n-2)])
```

2. Si A est une matrice (n,n), `diag(A)` renvoie un vecteur colonne contenant les éléments diagonaux de A (donc un vecteur colonne de dimension n). `diag(diag(A)` renvoie alors une matrice carrée diagonale d'ordre n, avec comme éléments diagonaux ceux de la matrice initiale.
3. Voici une possibilité :

```
--> A = rand(5,5)
--> T = tril(A) - diag(diag(A)) + eye(A)
```

4. (a)

```
--> Y = 2*X.^2 - 3*X + ones(X)
--> Y = 2*X.^2 - 3*X + 1   // en utilisant un raccourci d'ecriture
--> Y = 1  + X.*(-3 + 2*X) // plus un schema a la Horner
```

(b)

```
--> Y = abs(1 + X.*(-3 + 2*X))
```

(c)

```
--> Y = (X - 1).*(X + 4)  // en utilisant un raccourci d'ecriture
```

(d)

```
--> Y = ones(X)./(ones(X) + X.^2)
--> Y = (1)./(1 + X.^2) // avec des raccourcis
```

5. Voici le script:

```
n = 101;                            // pour  la discretisation
x = linspace(0,4*%pi,n);
y  =  [1  , sin(x(2:n))./x(2:n)]; // pour eviter la division par zero...
plot(x,y,"x","y","y=sin(x)/x")
```

6. Pour cet exercice, on suppose que l'on vient de rentrer dans Scilab[1] (avec la commande `clear`, on peut s'apercevoir que l'on gagne un peu de place car on détruit quelques variables chargées dans l'environnement initial). Si l'on tape who on obtient à la fin l'information :

```
 using 3837 elements out of 1000000.  and 41 variables out of 499
```

On peut calculer alors :

```
-->floor(sqrt(1000000 - 3837))
ans =

    998.
```

1. cette correction a été faite avec la version 2.3.1

et essayer de créer une matrice d'ordre 998

```
-->A = eye(998,998);
```

On retape `who` :

```
  using 999846 elements out of 1000000.  and 43 variables out of 499
```

Faisons les comptes :

```
-->  998*998 + 3837 +  2 + 2  + 1
-->  //  place pour le  contenu  de A + place utilisee initialement
-->  //  +  variable A +   variable  ans + place   pour le contenu de ans
  ans =

    999846.
```

Remarque : on ne peut alors quasiment plus travailler (il ne reste plus beaucoup de place). Pour augmenter la taille de la pile, il faut utiliser la commande stacksize (par exemple stacksize(2000000) pour doubler la taille par defaut).

Annexe B

Correction des exercices du chapitre 3

1. L'algorithme classique a deux boucles :

```
function  [x] =  sol_tri_sup1(U,b)
  //
  //  resolution  de  Ux  =   b ou U   est triangulaire superieure
  //
  //  Remarque : Cet algo fonctionne en cas de seconds membres multiples
  // (chaque second membre correspondant a une  colonne de b)
  //
  [n,m] =  size(U)
  // quelques   verifications ....
  if  n  ~=  m then
     error('  La matrice n''est pas  carree')
  end
  [p,q] =  size(b)
  if p ~=  m then
     error(' Second membre incompatible')
  end
  // debut de l'algo
  x = zeros(b) // on reserve de la place pour  x
  for i = n:-1:1
     somme = b(i,:)
     for j =  i+1:n
        somme = somme -  U(i,j)*x(j,:)
     end
     if U(i,i) ~= 0 then
        x(i,:) = somme/U(i,i)
     else
        error(' Matrice non inversible')
     end
  end
```

Voici une version utilisant une seule boucle

```
function [x] = sol_tri_sup2(U,b)
  //
  //  idem a sol_tri_sup1 sauf que l'on utilise un peu
  //  plus   la notation  matricielle
  //
  [n,m] = size(U)
  //  quelques verifications ....
  if  n ~= m then
     error('  La matrice n''est pas carree')
  end
```



```
  [p,q] = size(b)
  if p ~= m then
     error(' Second membre incompatible')
  end
  // debut de l'algo
  x = zeros(b) // on reserve de la place  pour x
  for i = n:-1:1
     somme = b(i,:) - U(i,i+1:n)*x(i+1:n,:) // voir le commentaire final
     if U(i,i) ~= 0 then
        x(i,:) = somme/U(i,i)
     else
        error(' Matrice non inversible')
     end
  end
```

Commentaire : lors de la premiere iteration (correspondant à $i = n$) les matrices `U(i,i+1:n)` et `x(i+1:n,:)` sont vides. Elles correspondent à un objet qui est bien defini en Scilab (la matrice vide) qui se note `[]`. L'addition avec une matrice vide est definie et donne : `A = A + []`. Donc lors de cette premiere iteration, on a `somme =  b(n,:) + []` c'est a dire que `somme = b(n,:)`.

2.
```
//
// script pour resoudre x" + alpha*x' + k*x = 0
//
// Pour mettre l'equation sous la forme d'un systeme du 1er ordre
// on pose  : X(1,t) =  x(t) et X(2,t) = x'(t)
//
// On obtient alors : X'(t) = A X(t) avec  :  A  = [0 1;-k -alpha]
//
k =  1;
alpha =  0.1;
T = 20;    //  instant final
n   = 100; //  discretisation temporelle :   l'intervalle  [0,T] va  etre
           //  decoupe  en n   intervalles
t  = linspace(0,T,n+1); // les  instants : X(:,i) correspondra a  X(:,t(i))
dt = T/n;  // pas de temps
A = [0 1;-k -alpha];
X = zeros(2,n+1);
X(:,1) = [1;1]; // les conditions  initiales
M = expm(A*dt); // calcul de  l'exponentielle de A dt

// le  calcul
for i=2:n+1
  X(:,i) =  M*X(:,i-1);
end

// affichage  des  resultats
xset("window",0)
xbasc()
xselect()
plot(t,X(1,:),'temps','position','Courbe x(t)')
xset("window",1)
xbasc()
xselect()
plot(X(1,:),X(2,:),'position','vitesse','Trajectoire dans le plan de phase')
```

3.
```
function [i,info]=intervalle_de(t,x)
  // recherche dichotomique de l'intervalle i tel que: x(i)<=  t <= x(i+1)
```

```
   // si t n'est pas dans [x(1),x(n)] on renvoie info = %f
   n=length(x)
   if (t<x(1)) | (t>x(n)) then
      info = %f
      i = 0 // on met une valeur par defaut
   else
      info = %t
      i_bas=1
      i_haut=n
      while  i_haut - i_bas > 1
         itest = floor((i_haut + i_bas)/2 )
         if ( t >= x(itest) ) then i_bas= itest, else, i_haut=itest, end
      end
      i=i_bas
   end
```

4. `function [p]=myhorner(t,x,c)`

```
   // evaluation  du polynome c(1) +  c(2)*(t-x(1)) +  c(3)*(t-x(1))*(t-x(2)) + ...
   // par l'algorithme d'horner
   // t est un vecteur d'instants (ou une matrice)

   n=length(c)
   p=c(n)*ones(t)
   for k=n-1:-1:1
      p=c(k)+(t-x(k)).*p
   end
```

5. Fabrication d'une serie de fourier tronquee :

```
function [y]=signal_fourier(t,T,cs)

   // cette  fonction renvoie un  signal T-periodique
   // t  : un vecteur d'instants pour lesquels on calcule
   //      le signal y ( y(i) correspond a t(i) )
   // T  : la periode du signal
   // cs : est un vecteur qui  donne l'amplitude  de chaque fonction f(i,t,T)

   l=length(cs)
   y=zeros(t)
   for j=1:l
      y=y + cs(j)*f(j,t,T)
   end

//------------------------------------------------------------------

function [y]=f(i,t,T)

   // les polynomes trigonometriques pour un signal de periode T :

   // si i est pair   : f(i)(t)=sin(2*pi*k*t/T)  (avec k=i/2)
   // si i est impair : f(i)(t)=cos(2*pi*k*t/T)  (avec k=floor(i/2))
   // d'ou en particulier  f(1)(t)=1 que l'on traite  ci dessous
   // comme un cas particulier bien que se ne soit pas necessaire
   // t est un vecteur d'instants

   if i==1 then
      y=ones(t)
```



```
   else
      k=floor(i/2)
      if modulo(i,2)==0 then
         y=sin(2*%pi*k*t/T)
      else
         y=cos(2*%pi*k*t/T)
      end
   end
```

6. Le produit vectoriel en « vectoriel »

```
function [v]=prod_vect_v(v1,v2)
   // le produit vectoriel en vectoriel...
   // v1 et v2 doivent etre des matrices (3,n)
   v=zeros(v1)
   v(1,:) = v1(2,:).*v2(3,:) - v1(3,:).*v2(2,:)
   v(2,:) = v2(1,:).*v1(3,:) - v2(3,:).*v1(1,:)
   v(3,:) = v1(1,:).*v2(2,:) - v1(2,:).*v2(1,:)
```

Bibliographie

[1] David GOLDBERG, *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, ACM Computing Surveys, Vol. 23, N. 1, March 1991.

[2] Donald KNUTH, *The Art of Computer Programming*, Addison-Wesley, Reading, Mass., vol II, 2nd ed.

[3] Reuven Y. RUBINSTEIN, *Simulation and the Monte Carlo Method*, John Wiley & Sons, Wiley Series in Probability and Mathematical Statistics.

[4] Ernst HAIRER, *Polycopié « Analyse Numérique »*, http://www.unige.ch/math/folks/hairer/polycop.html

Index