Une introduction à Scilab

version 0.999

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 décrit une petite partie des possibilités de Scilab, essentiellement celles qui permettent la mise en pratique de notions d'analyse numérique et de petites simulations stochastiques, c'est à dire :

- la manipulation des matrices et vecteurs de nombres flottants ;
- la programmation en Scilab ;
- quelques primitives graphiques ;
- quelques fonctions importantes pour ces deux domaines (génération de nombres aléatoires, résolution d'équations, ...).

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.

Pour cette nouvelle version (je n'avais pas touché à ce document depuis 3 ans ...) j'ai essentiellement remanié le chapitre sur le graphique. A force de rajouter quelques paragraphes ici et là, ce document n'est plus très synthétique mais il existe maintenant d'autres introductions que vous pouvez récupérer à partir du site Scilab (voir plus loin). Finalement ce tutoriel est relatif à la version 2.7 de Scilab mais presque tous les exemples doivent fonctionner avec Scilab-2.6.

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) ;
- à mes collègues et amis, Stéphane Mottelet[1] , Antoine Grall, Christine Bernier-Katzentsev et Didier Schmitt ;
- un grand merci à Patrice Moreaux pour sa relecture attentive et les corrections dont il m'a fait part ;
- à Helmut Jarausch, qui a traduit ce document en allemand, et qui m'a signalé quelques erreurs supplémentaires ;
- et à tous les lecteurs qui m'ont apporté leurs encouragements, remarques et corrections.

[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 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

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

« 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 2.7 du logiciel. Pour tout renseignement consulter la « Scilab home page » :

`http ://scilabsoft.inria.fr`

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

Le « Scilab Group » a écrit (entre fin 1999 et 2001) une vingtaine d'articles dans la revue « Linux magazine ». Plusieurs aspects de Scilab (dont la plupart ne sont pas évoqués dans cette introduction) y sont présentés, je vous les recommande donc. Ces articles sont consultables à partir de l'url :

`http ://www.saphir-control.fr/articles/`

Scilab dispose aussi d'un forum usenet qui est le lieu adéquat pour poser des questions, faire des remarques, 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 en cliquant sur l'item Scilab newsgroup archive[2].

Toujours à partir de la page Scilab, vous avez accès a un certain nombre de documents en choisissant l'une des deux rubriques Books and Articles on Scilab ou Scilab Related Links. En particulier :

- l'introduction de B. Ycart (Démarrer en Scilab) ;
- « Scilab Bag Of Tricks » de Lydia E. van Dijk et Christoph L. Spiel qui est plutôt destiné aux personnes connaissant déjà bien Scilab (le développement de ce livre s'est hélas arrêté brutalement il y a quelques années) ;
- Travaux Pratiques sur Scilab classes par themes vous permet d'accéder à des projets réalisés avec Scilab par des élèves de l'ENPC ;
- une introduction à l'informatique en utilisant Scilab (`http ://kiwi.emse.fr/SCILAB/`).

Mais il y en a bien d'autres, et, selon vos besoins vous trouverez sans doute des documents plus adaptés que cette introduction.

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

[2] il s'agit en fait, d'un lien sur l'archivage opéré par Google.

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 :

```
scilab
```

dans un terminal[1]. Si tout se passe bien, la fenêtre Scilab apparaît à l'écran avec en haut un menu (donnant en particulier accès au `Help`, aux `Demos`) suivi de la bannière scilab et de l'invite (`-->`) qui attend vos commandes :

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



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 :

[1] Ou de cliquer sur un item de menu ou une icône prévus pour cet effet !

```
 A  =
!   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`) :

matrices identités

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 $\mathbb{K}^n$.)

matrices diagonales, extraction de la diagonale

Pour obtenir une matrice diagonale, dont les éléments diagonaux sont formés à partir 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).

matrices de zéros et de uns

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. !
```

extractions des parties triangulaires supérieure et inférieure

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. !
```

matrices de nombres aléatoires

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 !
```

vecteurs à incrément constant entre 2 composantes

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 $(n, m) \times (m, p) \ldots$), 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 !

abs	valeur absolue ou module
exp	exponentielle
log	logarithme népérien
log10	logarithme base 10
cos	cosinus (argument en radian)
sin	sinus (argument en radian)
sinc	$\frac{sin(x)}{x}$
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

```
-->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 ?
      !--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 :

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

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.

A propos de l'ordre des éléments d'une matrice

Les matrices de Scilab sont stockées colonne par colonne et cet ordre des éléments intervient dans plusieurs fonctions (voir par exemple `matrix` qui permet de reformatter une matrice). En particulier, pour les opérations d'extraction et d'insertion, il est possible d'utiliser cet ordre implicite en utilisant seulement un seul vecteur d'indice (au lieu de deux, l'un désignant les colonnes et l'autre les lignes). Voici quelques exemples à partir de la matrice A précédente :

```
-->A(5)
 ans  =
    2.

-->A(5:9)
 ans  =
!   2.  !
!   4.  !
!   8.  !
!   16. !
!   3.  !
```

```
-->A(5:9) = -1  // il s'agit d'une insertion
 A  =
!   1.  - 1.  - 1.     4.   !
!   1.  - 1.    9.     16.  !
!   1.  - 1.    27.    64.  !
!   1.  - 1.    81.    256. !
```

2.4 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`.

2.5 Utilisation de l'aide en ligne

Elle s'obtient en cliquant sur le bouton `Help` de la fenêtre Scilab... Depuis la version 2.7 les pages d'aides sont au format html[2]. Par défaut un navigateur assez primitif (son rendu html est peu réjouissant) est utilisé mais vous pouvez choisir un autre programme avec l'astuce suivante :

```
-->global %browsehelp; %browsehelp=[];
-->help
```

La première ligne modifie la variable globale `%browsehelp` qui contient alors la matrice vide. Lors de l'entrée de la commande help un menu vous propose un choix entre différentes possibilités (par exemple sous Unix on peut choisir mozilla). En fait vous pouvez positionner cette variable dans votre fichier `.scilab` ce qui vous évitera cette manipulation[3].

Si vous rentrez simplement la commande `help` (ou si vous cliquez sur le bouton `Help`) alors la page html affichée correspond à un classement de toutes les pages d'aide en un certain nombre de rubriques (`Scilab Programming, Graphic Library, Utilities and Elementary functions,...`). En cliquant sur une rubrique particulière vous obtenez la liste de toutes les fonctions classées dans cette rubrique, chacune étant accompagnée d'une brève description. En cliquant alors sur l'une de ces fonctions vous obtenez la page de la fonction en question.

Si vous connaissez le nom de la fonction qui vous intéresse, vous pouvez rentrer la commande `help` *nom_de_la_fonction* dans la fenêtre de commande Scilab pour obtenir la page directement.

Enfin la commande `apropos` *mot_clé* vous permet d'obtenir toutes les pages dans lesquelles la chaîne de caractères *mot_clé* apparaît dans le nom de la fonction ou dans sa brève description.

Actuellement il peut être difficile de se diriger en utilisant les intitulés des rubriques (par exemple Elementary functions est un ensemble « fourre-tout » qui mériterait d'être redécoupé), n'hésitez donc pas à utiliser `apropos`.

2.6 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.

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] le format de base est en fait du xml à partir duquel on obtient le html.

[3] si votre navigateur favori n'est pas prévu dans le choix proposé par défaut il faut aller modifier le fichier `SCI/macros/util/browsehelp.sci`.

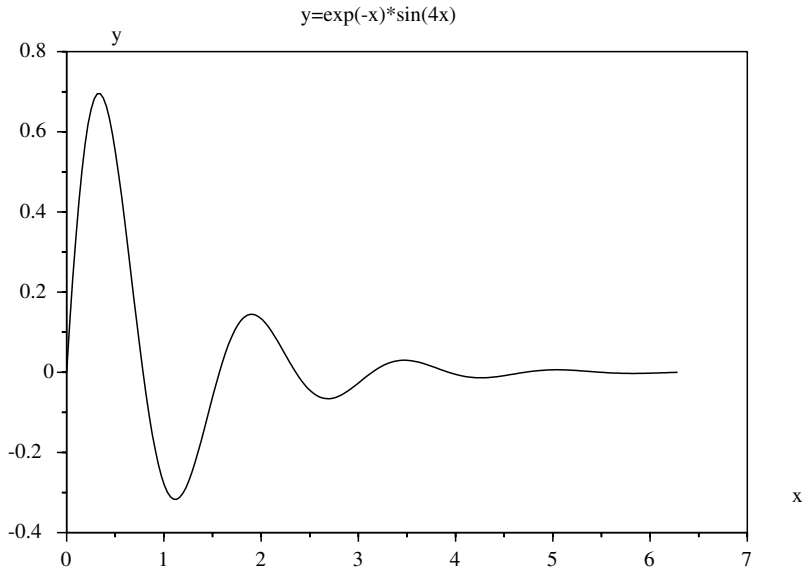

FIG. 2.1 – Un graphe simple

2.7 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. certains éditeurs peuvent être munis d'un mode d'édition spécifique pour Scilab (voir la page Scilab). Pour `emacs` il en existe deux mais le meilleur est de loin celui d'Alexander Vigodner dont les dernières versions sont téléchargeables à partir de l'url :

3. un script sert souvent de programme principal d'une application écrite en Scilab.

2.8 Compléments divers

2.8.1 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)  ou  1.0./f(x)
```

2.8.2 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[4], 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[5]. 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 !
```

[4]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

[5]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`).

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[6]. 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 =
    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.

2.8.3 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,"r")  // effectue la somme de chaque colonne -> on obtient une ligne
 ans  =
!   5.    7.    9. !
```

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

```
-->sum(B,"c") // 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 :

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

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,"r")  // taper B pour revoir cette matrice...
 ans  =
!   4.    10.    18. !

-->prod(B,"c")
 ans  =
!   6.   !
!   120. !

-->prod(B)
 ans  =
    720.

-->prod([])
 ans  =
    1.
```

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

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

somme et produit cumulés

Les fonctions `cumsum` et `cumprod` calculent respectivement les sommes et produits cumulés d'un vecteur ou d'une matrice :

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

-->cumsum(x)
 ans  =
!   1.    3.    6.    10.    15.    21. !

-->cumprod(x)
 ans  =
!   1.    2.    6.    24.    120.    720. !
```

Pour une matrice l'accumulation se fait simplement selon l'ordre colonne par colonne :

```
-->x = [1 2 3;4 5 6]
 x  =
!   1.    2.    3. !
!   4.    5.    6. !

-->cumsum(x)
 ans  =
!   1.    7.     15. !
!   5.    12.    21. !
```

Et comme pour les fonctions `sum` et `prod`, on peut aussi faire les sommes et produits cumulés selon les lignes et les colonnes :

```
-->cumsum(x,"r")   // produit cumule selon les colonnes !
 ans  =
!   1.    2.    3. !
!   5.    7.    9. !

-->cumsum(x,"c")   // produit cumule selon les lignes !
 ans  =
!   1.    3.    6.  !
!   4.    9.    15. !
```

Ici apparaît la maladresse du choix de Scilab pour les options restreingnant certaines opérations à s'effectuer selon les lignes ou les colonnes[7] : pour les fonctions `sum` et `prod` elles apparaissent comme désignant la forme finale obtenue (`sum(x,"r")` permet d'obtenir un vecteur ligne (r pour row, ligne en anglais) c'est donc la somme de chaque colonne qui est effectuée) mais ce moyen mnémotechnique ne marche plus pour ces fonctions !

minimum et maximum d'un vecteur ou d'une matrice

Les fonctions `min` et `max` se chargent de ces opérations. Elles fonctionnent exactement comme `sum` et `prod` quant à l'argument supplémentaire pour calculer les minima ou maxima de chaque ligne ou colonne. Elles admettent aussi un argument de sortie supplémentaire donnant le premier indice où le minimum ou maximum est atteint (les premiers indices pour les minima maxima selon les lignes ou les colonnes). Exemples :

[7] `sum`, `prod`, `cumsum`, `cumprod`, `mean`, `st_deviation`, `min`, `max`

```
-->x = rand(1,5)
 x  =
!   0.7738714    0.7888738    0.3247241    0.4342711    0.2505764 !

-->min(x)
 ans  =
    0.2505764

-->[xmin, imin] = min(x)
 imin  =
    5.              // le min est obtenu en x(5)
 xmin  =
    0.2505764

-->y = rand(2,3)
 y  =
!   0.1493971    0.475374     0.8269121 !
!   0.1849924    0.1413027    0.7530783 !

-->[ymin, imin] = min(y)
 imin  =
!   2.    2. !      // le min est obtenu pour y(2,2)
 ymin  =
    0.1413027

-->[ymin, imin] = min(y,"r")  // minimum de chaque colonne
 imin  =
!   1.    2.    2. !          // => les min sont  y(1,1) y(2,2) y(2,3)
 ymin  =
!   0.1493971    0.1413027    0.7530783 !

-->[ymin, imin] = min(y,"c")  // minimum de chaque ligne
 imin  =
!   1. !           // les min sont  y(1,1)
!   2. !           //               y(2,2)
 ymin  =
!   0.1493971 !
!   0.1413027 !
```

moyenne et écart type

Les fonctions `mean` et `st_deviation` permettent de calculer la moyenne et l'écart type des composantes d'un vecteur ou d'une matrice. La formule utilisée pour l'écart type étant :

$$\sigma(x) = \left(\frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2 \right)^{1/2} , \; \bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$$

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

-->mean(x)
 ans  =
    3.5
```

```
-->st_deviation(x)
 ans  =
    1.8708287
```

De même on peut calculer la moyenne (et aussi l'écart type) des lignes ou des colonnes d'une matrice :

```
-->x = [1 2 3;4 5 6]
 x  =
!   1.    2.    3. !
!   4.    5.    6. !

-->mean(x,"r")   // les moyennes de chaque colonne
 ans  =

!   2.5    3.5    4.5 !

-->mean(x,"c")   // les moyennes de chaque ligne
 ans  =

!   2. !
!   5. !
```

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 = [1 2 3; 4 5 6]
 B  =
!   1.    2.    3. !
!   4.    5.    6. !

-->B_new = matrix(B,3,2)
 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`.

2.8.4 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.

2.9 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. La fonction `tril` permet d'extraire la partie triangulaire inférieure d'une matrice (`triu` pour la partie triangulaire supérieure). 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. *Une petite illustration de la loi des grands nombres :* obtenir avec la fonction `rand` n réalisations de la loi uniforme sous la forme d'un vecteur x, calculer la moyenne cumulée de ce vecteur, c-a-d le vecteur $\bar{x}$ dont les n composantes sont : $\bar{x}_k = \frac{1}{k}\sum_{i=1}^{k} x_i$ et tracer la courbe de la suite ainsi obtenue. Tester avec des valeurs de n de plus en plus grandes.

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
```

[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`

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.

3.2 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
```

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

```
    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
   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. !
```

[5] les entiers étant vu comme des nombres flottants

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 =
 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 $2^{\text{ème}}$ un vecteur de chaînes de caractères, et le $3^{\text{ème}}$ 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 sommets par face,nb de faces)) 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.

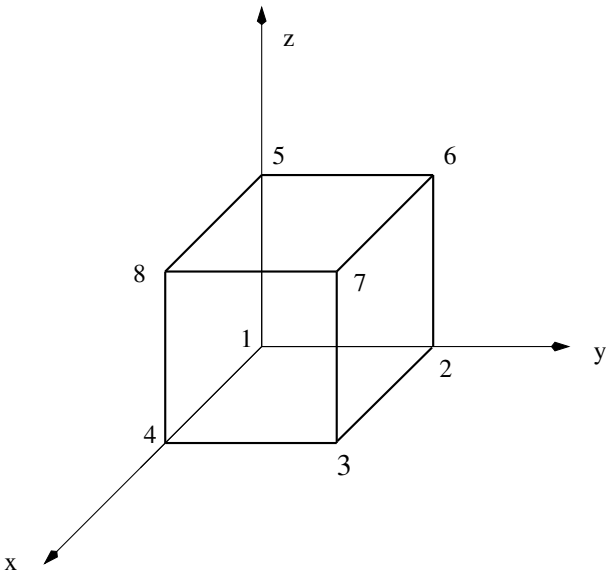

FIG. 3.1 – Numéros des sommets du cube

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 5 3 2 1 1;...  // la connectivité des faces
           2 8 7 6 5 4;...
           3 7 8 7 6 8;...
           4 6 4 3 2 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.    5.    3.    2.    1.    1. !
!   2.    8.    7.    6.    5.    4. !
!   3.    7.    8.    7.    6.    8. !
!   4.    6.    4.    3.    2.    5. !
```

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

```
-->Cube.coord(:,2)  // ou encore  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. Vient ensuite le corps de la fonction (qui permet de calculer les arguments de sortie à partir des arguments d'entrée). La fonction doit se terminer par le mot clé `endfunction`. *Rmq :* 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)
endfunction
```

Supposons que l'on ait écrit cette fonction dans le fichier `facts.sci`[6]. Pour que Scilab puisse la connaître, il faut charger le fichier par l'instruction :

```
getf("facts.sci")  // ou encore exec("facts.sci")
```

ce qui peut aussi se faire via le menu `File operations` comme pour les scripts. 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 d'autres exemples, 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).

Comme deuxième exemple, prenons la résolution d'une équation du second degré :

```
function [x1,x2] = resoud_equ_2d(a, b, c)
   // calcul des racines de a x^2 + b x + c = 0
   // a, b et c peuvent etre des reels ou des complexes et a doit etre non nul
   delta = b^2 - 4*a*c
   x1 = (-b - sqrt(delta))/(2*a)
   x2 = (-b + sqrt(delta))/(2*a)
endfunction
```

Voici trois essais avec cette fonction :

```
-->[r1, r2] = resoud_equ_2d(1,2,1)
 r2  =
  - 1.
 r1  =
  - 1.

-->[r1, r2] = resoud_equ_2d(1,0,1)
 r2  =
```

[6]La tradition est de suffixer les noms des fichiers contenant des fonctions en `.sci` (et en `.sce` pour les scripts)

```
    i
 r1  =
  - i

-->resoud_equ_2d(1,0,1)  // appel sans affectation
 ans  =
  - i
```

On peut remarquer que le troisième appel ne renvoie qu'une seule racine (la première). Ceci est normal car on n'a pas affecté le resultat de la fonction (qui renvoie 2 objets scalaires) contrairement aux deux premiers appels. Dans ce cas scilab utilise la variable par défaut `ans` pour stocker le résultat mais elle ne peut que stocker qu'un le premier objet renvoyé par la fonction (le deuxième étant perdu).

Voici un troisième 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
endfunction
```

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 (*)

Depuis la version 2.4 de Scilab, le passage d'un paramètre se fait par référence si ce paramètre n'est pas modifié par la fonction et par copie sinon (ainsi les paramètres d'entrée ne peuvent pas être modifiés). Vous pouvez en tenir compte pour accélerer certaines fonctions. Voici un exemple caricatural mais qui met bien en évidence le coût du passage par copie pour des arguments prenant beaucoup de place :

```
function [w] = toto(v, k)
  w = 2*v(k)
endfunction

function [w] = titi(v, k)
  v(k) = 2*v(k)
  w = v(k)
endfunction

// script de test
m = 200000;
nb_rep = 2000;
x = rand(m,1);
timer(); for i=1:nb_rep; y = toto(x,300); end; t1=timer()/nb_rep
timer(); for i=1:nb_rep; y = titi(x,300); end; t2=timer()/nb_rep
```

Sur ma machine j'obtiens :

```
 t1  =  0.00002
 t2  =  0.00380
```

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

3.4.2 Déverminage d'une fonction

Pour déboguer une fonction vous pouvez utiliser la fonction `disp(v1,v2, ...)` qui permet de visualiser la valeur des variables `v1`, `v2`,...Attention `disp` affiche ses arguments dans l'ordre inverse !

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`).

Comme il est pénible de rajouter des instructions `pause` dans toutes les fonctions à déboguer, il existe un moyen de mettre des points d'arrêt (break points en anglais) avec l'instruction :

`setbpt(`*nom_fonction* `[`, *num_ligne* `])`

où vous entrez le nom de la fonction comme une chaîne de caractères et (éventuellement) le numéro de la ligne d'arrêt (la valeur par défaut est 1). Lorsque Scilab rencontre un tel point d'arrêt tout se passe comme si vous aviez mis une instruction `pause` *après* la ligne mentionnée. Après examen des variables, vous repartez donc avec un `resume`.

Les points d'arrêt doivent être explicitement enlevés avec :

`delbpt(`*nom_fonction* `[`, *num_ligne* `])`

Si vous ne précisez pas le numéro de ligne, tous les points d'arrêt installés dans cette fonction sont détruits. Enfin, vous pouvez voir tous les points d'arrêt que vous avez définis avec `dispbpt()`.

Voici un petit exemple avec la fonction `resoud_equ_2d` définie précédemment :

```
function [x1,x2] = resoud_equ_2d(a, b, c)
   // calcul des racines de a x^2 + b x + c = 0
   // a, b et c peuvent etre des reels ou des complexes et a doit etre non nul
   delta = b^2 - 4*a*c
   x1 = (-b - sqrt(delta))/(2*a)
```

```
    x2 = (-b + sqrt(delta))/(2*a)
  endfunction
```

Je suppose que cette fonction a été chargée dans Scilab, soit avec un `getf` soit avec un `exec` :

```
-->setbpt("resoud_equ_2d")   // un premier point d'arrêt
-->[r1,r2] = resoud_equ_2d(1,2,7)  // je lance le calcul => arrêt
Stop after row     1 in function resoud_equ_2d :

-1->a    // j'examine quelques variables
 a  =
    1.

-1->b
 b  =
    2.

-1->dispbpt()   // je visualise mes points d'arrêt
 breakpoints of  function :resoud_equ_2d
          1

-1->setbpt("resoud_equ_2d",5)  // je rajoute un point d'arrêt en ligne 5

-1->x1  // x1 n'est pas encore definie !
x1
  !--error     4
undefined variable : x1

-1->resume   // je redémarre (jusqu'au prochain point d'arrêt)
Stop after row     5 in function resoud_equ_2d :

-1->x1     // je peux maintenant examiner x1
 x1  =
   - 1. - 2.4494897i

-1->x2    // x2 n'est tj pas définie (x2 est définie à la ligne 6 de cette fct)
x2
  !--error     4
undefined variable : x2

-1->dispbpt()   // je vois mes 2 points d'arrêt
 breakpoints of  function :resoud_equ_2d
         1
         5

-1->resume   // je redémarre (et donc je sorts de ma fonction)
 r2  =
   - 1. + 2.4494897i
 r1  =
   - 1. - 2.4494897i

-->delbpt("resoud_equ_2d")  // je détruits tous les points d'arrêt
```

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[7]. 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

[7] si la boucle est imbriquée dans une autre, `break` permet de sortir uniquement de la boucle interne

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
endfunction
```

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

Rappel : on peut remarquer l'emploi de la fonction `size` alors que la fonction `length` semble plus adaptée pour un vecteur[8] ; 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).

[8] 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))`

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)
endfunction
```

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)
endfunction
```

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

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)
endfunction
```

La fonction argn et les arguments optionnels

`argn` 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()
```

`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).

Cependant pour écrire aisément des fonctions avec des arguments optionnels, Scilab possède une fonctionnalité très pratique d'association entre arguments formels et arguments effectifs. Voici un exemple de fonction pour laquelle je désire un argument classique plus deux paramètres optionnels :

```
function [y] = argopt(x, coef_mult, coef_add)
   // pour illustrer l'association entre argument formels et effectifs
   [lhs, rhs] = argn()
   if rhs < 1 | rhs > 3 then
      error("mauvais nombre d''arguments")
   end

   if ~exists("coef_mult","local") then
      coef_mult = 1
   end

   if ~exists("coef_add","local") then
      coef_add = 0
   end

   y = coef_mult*x + coef_add
endfunction
```

La fonction `exists` me permet de tester si les arguments `coef_mult` et `coef_add` ont été définis[9] lors de l'appel de la fonction, ce qui permet de leur donner une valeur par défaut dans le cas contraire. De plus avec la syntaxe *argument_formel = argument_effectif*, on peut se permettre de mettre les arguments dans n'importe quel ordre. Exemples :

```
-->y1 = argopt(5)
 y1  =
    5.

-->y2 = argopt(5, coef_add=10)
 y2  =
    15.

-->y3 = argopt(5, coef_mult=2, coef_add=6)
 y3  =
    16.

-->y4 = argopt(5, coef_add=6, coef_mult=2) // y4 doit être égal à y3
 y4  =
    16.
```

3.5 Compléments divers

3.5.1 Longueur des identificateurs

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

```
-->a2345678901234567890123456 = 1
 a234567890123456789012345  =
    1.

-->a2345678901234567890123456
 a234567890123456789012345  =
    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) :

[9] l'argument supplémentaire avec la valeur `"local"` réduit la portée de l'interrogation à la fonction elle même ; ceci est important car des variables de même nom peuvent être définies à des niveaux supérieurs.

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

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

   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
endfunction

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
endfunction
```

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))
endfunction

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)
endfunction
```

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)
```

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 ; **cependant depuis la version 2.6 vous pouvez définir des fonctions dans un script** avec la syntaxe habituelle ce qui est plus pratique qu'avec un `deff` ; dans votre script vous pouvez donc écrire les fonctions au début du texte :

   ```
   // definition des fcts utilisées par le script
   function
      ....
   endfunction
   function
      ....
   endfunction
   // debut effectif du script
   ....
   ```

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.

[11] voir la section « Primitives et fonctions Scilab » du bétisier

[12] qui est inutile en dehors de son aspect pédagogique : cf `fplot2d`

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 !
```

3.6 Lecture/écriture sur fichiers ou dans la fenètre Scilab

Scilab possède deux familles d'entrées/sorties. Il faut éviter de mélanger les instructions de ces deux familles surtout si vous écrivez/lisez dans un fichier.

[13] et aussi sur une liste, voir le Help

3.6.1 Les entrées/sorties à la fortran

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.

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 :

[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`

```
-->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
.......
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
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
```

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

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)).

3.6.2 Les entrées/sorties à la C

Pour l'ouverture et la fermeture des fichiers, il faut utiliser `mopen` et `mclose`, les instructions de base sont :

`mprintf, mscanf`	ecriture, lecture dans la fenêtre scilab
`mfprintf, mfscanf`	ecriture, lecture dans un fichier
`msprintf, msscanf`	ecriture, lecture dans une chaîne de caractères

Dans la suite, j'explique uniquement l'usage de `mprintf`. Voici un script pour vous décrire les cas principaux :

```
n = 17;
m = -23;
a = 0.2;
b = 1.23e-02;
c = a + %i*b;
s = "coucou";
mprintf("\n\r  n = %d, m = %d", n, m); // %d pour les entiers
mprintf("\n\r  a = %g", a);            // %g pour les reels
mprintf("\n\r  a = %e", a);            // %e pour les reels (notation avec exposant)
mprintf("\n\r  a = %24.16e", a);       // on impose le nombre de "décimales"
mprintf("\n\r  c = %g + i %g", real(c), imag(c)); // nb complexe
mprintf("\n\r  s = %s", s);            // %s pour les chaines de caractères
```

Si vous executez ce script avec un « `;` » à la fin de l'ordre `exec` (c-a-d `exec("demo_mprintf.sce") ;` si vous avez appelé le fichier par ce nom), alors vous allez observer la sortie suivante :

```
n = 17, m = -23
a = 0.2
a = 2.000000e-01
a =   2.0000000000000001e-01
c = 0.2 + i 0.0123
s = coucou
```

Quelques explications :

- le `\n\r` impose un passage à la ligne (sous Unix le `\n` doit suffire) : ne pas le mettre si vous ne voulez pas aller à la ligne !
- les `%`x sont des directives de formatage, c'est à dire qu'elles donnent la façon dont la variable (ou constante) doit être écrite :
  1. `%d` pour les entiers ;
  2. `%e` pour les réels en notation scientifique (c-a-d avec exposant). La sortie doit prendre la forme : $[-]c_0.c_1...c_6\mathrm{e}\pm d_1d_2[d_3]$ c'est à dire avec le signe (si le nombre est négatif), 1 chiffre avant la virgule (le point en fait !), 6 chiffres après, la lettre e puis le signe de l'exposant, et enfin l'exposant sur deux chiffres, voire 3 si besoin. Pour imposer plus précisemment la sortie, on rajoute deux nombres entiers séparés par un point, le premier donnant le nombre de caractères total et le deuxième, le nombre de chiffres après la virgule.
  3. `%g` permet de choisir entre une sortie sans exposant (si c'est possible) ou avec.
  4. `%5.3f` permet d'avoir une sortie sans exposant avec 5 caractères en tout dont 3 après la virgule.
  5. `%s` s'utilise pour les chaînes de caractères.

– à chaque directive doit correspondre une valeur de sortie (variable, expression ou constante), par exemple, si on veut afficher 4 valeurs, il faut 4 directives :

```
mprintf("   %d1 ..... %d4 ", expr1, expr2, expr3, expr4);
```

Question : la sortie de 0.2 avec le format `%24.16e` paraît bizarre. Réponse : c'est parce que 0.2 ne tombe pas juste en binaire et donc en mettant suffisamment de précision (pour la conversion binaire vers décimal) on finit par détecter ce problème.

3.7 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
endfunction
```

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
endfunction
```

il n'y a plus ce problème et l'on gagne un peu de temps : :

```
-->t = linspace(0,1,1000)';
-->timer(); A = vandm1(t,200); timer()
 ans  =
    6.04

-->timer(); A = vandm2(t,200); timer()
 ans  =
    1.26
```

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
endfunction

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
endfunction
```

et on améliore ainsi la rapidité de façon significative :

```
-->timer(); A = vandm5(t,200); timer()
 ans  =
     0.05

-->timer(); A = vandm6(t,200); timer()
 ans  =
     0.02
```

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

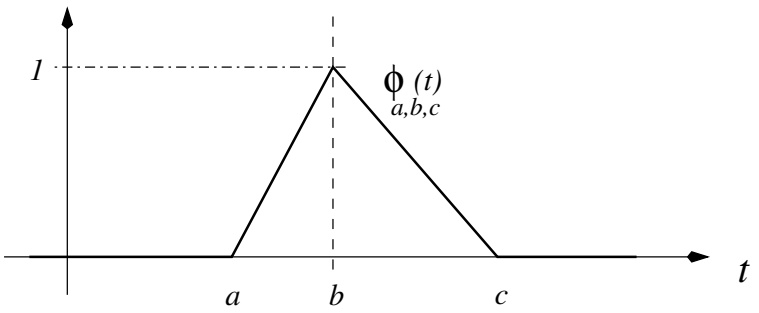

FIG. 3.2 – Fonction « chapeau »

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
endfunction
```

donne le résultat :

```
-->a = -0.2 ; b=0 ; c=0.15;
-->t = rand(200000,1)-0.5;
-->timer(); y1 = phi1(t,a,b,c); timer()
 ans  =
    2.46
```

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 \og element par element \fg.
   // 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)
endfunction

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)
endfunction
```

sont plus rapides :

```
-->timer(); y2 = phi2(t,a,b,c); timer()
 ans  =
    0.12

-->timer(); y3 = phi3(t,a,b,c); timer()
 ans  =
    0.12

-->timer(); y4 = phi4(t,a,b,c); timer() // voir plus loin la définition de phi4
 ans  =
    0.1
```

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

Remarques :

- ma petite optimisation pour `phi2` ne donne aucun gain (alors que c'était le cas pour une version précédente de scilab sur une autre machine) ;
- le code de `phi4` utilise la fonction `find` qui est sans doute 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. !
```

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 le code de la fonction `phi4` :

```
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)
endfunction
```

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.

3.8 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 1ère 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 $x(t_1), x(t_2), \ldots, x(t_n)$. 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)
endfunction
```

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.

7. *La rencontre* : Mr A et Mlle B ont décidé de se donner rendez-vous entre 17 et 18h. Chacun arrivera au hasard, uniformément et indépendamment l'un de l'autre, dans l'intervalle $[17, 18]$. Mlle B attendra 5 minutes avant de partir, Mr A 10 minutes.
   (a) Quelle est la probabilité qu'ils se rencontrent ? (rep 67/288)
   (b) Retrouver (approximativement) ce résultat par simulation : écrire une fonction `[p] = rdv(m)` renvoyant la probabilité empirique de la rencontre obtenue avec m réalisations.
   (c) Expérimenter votre fonction avec des valeurs de m de plus en plus grande.

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 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
```

Continuons avec une troisième courbe qui ne tient pas dans l'échelle[4] précédente :

```
yter = 2*y;
plot2d(x,yter)
```

on remarque que Scilab a changé l'échelle pour s'adapter à la troisième courbe[5] mais aussi qu'il a redessiné les deux premières courbes en fonction de la nouvelle échelle : ceci semble très naturel mais ce mécanisme n'est présent que depuis la version 2.6 (auparavant les deux premières courbes n'étaient pas redessinées et dans notre cas les courbes 2 et 3 étaient confondues !).

Il est possible d'afficher les 3 courbes simultannément :

```
xbasc() // pour effacer
plot2d(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.

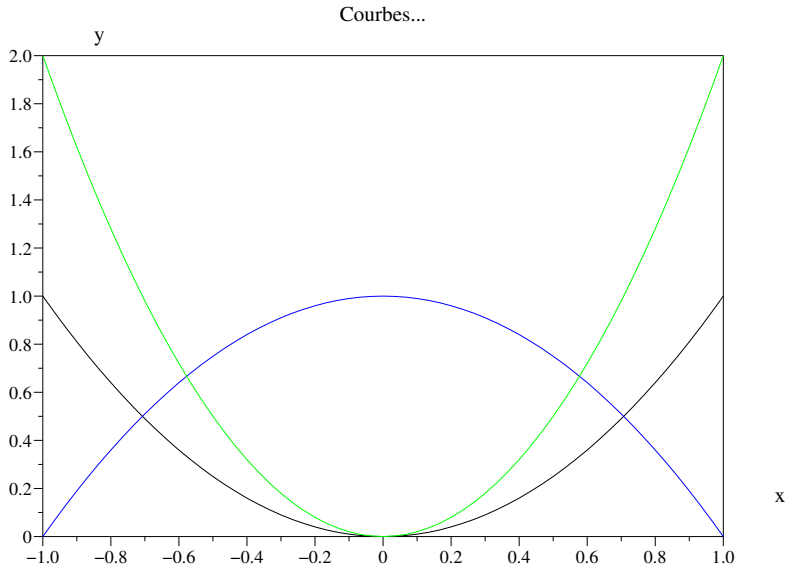

FIG. 4.1 – Les fonctions x^2, $1 - x^2$ et $2x^2$

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). Dans le cas où les vecteurs d'abscisses sont les mêmes (comme dans mon exemple) on peut simplement donner ce vecteur au lieu de le concaténer `nc` fois (`plot2d(x,My)` au lieu de `plot2d([x x .. x],My)`).

[4] Par échelle on sous-entend le rectangle de visualisation et éventuellement des propriétés supplémentaires.

[5] En fait l'échelle s'adapte pour que toutes les courbes puissent être visualisées complètement.

4.3 plot2d avec des arguments optionnels

La syntaxe générale a la forme :

`plot2d(Mx,My <,opt_arg>*)`

où `<,opt_arg>*` désigne l'éventuelle séquence des arguments optionnels `opt_arg` prenant la forme[6] :

mot_clé=valeur

la séquence pouvant être mise dans n'importe quel ordre. Nous allons expérimenter les principales options à travers plusieurs exemples :

1. **choisir les couleurs et définir la légende** : dans l'exemple précédent vous avez pu remarquer que Scilab a tracé les 3 courbes avec 3 couleurs différentes. Il a en fait utilisé les couleurs 1, 2 et 3 de la carte des couleurs par défaut. En voici d'autres :

1	noir	5	rouge vif	23	violet
2	bleu	6	mauve	26	marron
3	vert clair	13	vert foncé	29	rose
4	cyan	16	bleu turquoise	32	jaune orangé

   mais l'instruction `xset()` vous les montrera toutes[7]. Pour choisir les couleurs, on utilise la combinaison *style=vect* où vect est un vecteur ligne donnant les numéros de couleur pour chaque courbe ; la légende s'obtient avec *leg=str* où str est une chaîne de caractères de la forme `"leg1@leg2@..."` leg_i étant la légende pour la i ème courbe. Voici un exemple (cf figure (4.2)) :

```
x = linspace(0,15,200)';
y = besselj(1:5,x);
xbasc()
plot2d(x, y, style=[2 3 4 5 6], leg="J1@J2@J3@J4@J5")
xtitle("Les fonctions de Bessel J1, J2,...","x","y")
```

   mais comme l'ordre des arguments optionnels n'a pas d'importance on aurait pu aussi bien utiliser :

```
plot2d(x, y, leg="J1@J2@J3@J4@J5", style=[2 3 4 5 6])
```

2. **dessiner avec des symboles** : dans certains contextes, on préfère dessiner une marque pour chaque point et ne pas relier les points par des segments de droite. Pour cela il faut choisir une valeur de style entre 0 et -9 qui vous permet d'obtenir l'un des symboles suivant :

style	0	-1	-2	-3	-4	-5	-6	-7	-8	-9
symbole	·	+	×	⊕	♦	◇	△	▽	♣	◯

   Essayez par exemple (cf figure (4.3)) :

```
x = linspace(0,2*%pi,40)';
y = sin(x);
yp = sin(x) + 0.1*rand(x,"normal"); // perturbation gaussienne
xbasc()
plot2d(x, [yp y], style=[-2 2], leg="y=sin(x)+perturbation@y=sin(x)")
```

3. **spécifier l'échelle** : voici un exemple où il est nécessaire d'imposer une échelle isométrique car on cherche à dessiner un cercle (cf figure (4.4)) ; la combinaison correspondante est *frameflag=val* où val doit prendre la valeur 4 pour obtenir une échelle isométrique (obtenue à partir des maxima et minima des données) :

[6] en fait *argument_formel=argument_effectif*

[7] la carte des couleurs par défaut est loin d'être optimale avec des couleurs très proches les unes des autres mais on apprendra plus loin à la modifier.

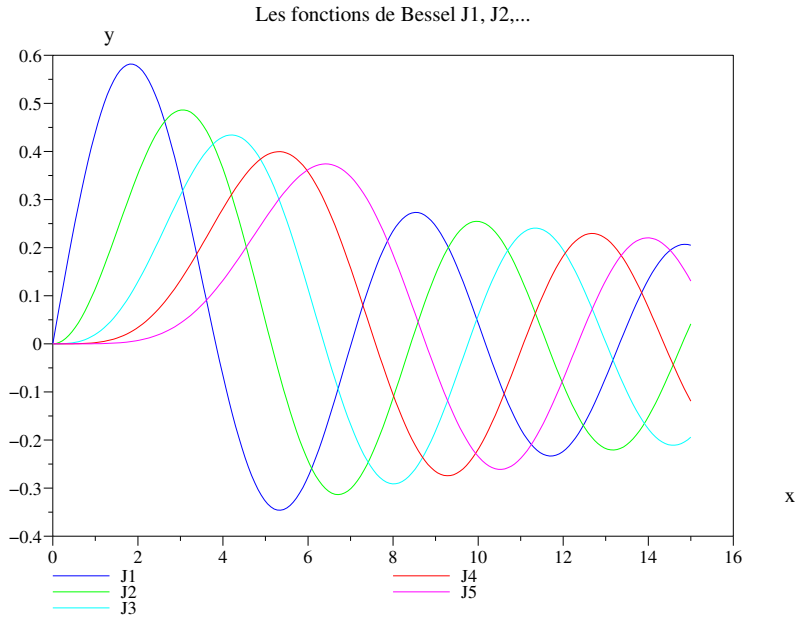

FIG. 4.2 – choix du style et de la légende

```
t = linspace(0,2*%pi,60)';
x1 = 2*cos(t); y1 = sin(t);            // une ellipse
x2 = cos(t);   y2 = y1;                // un cercle
x3 = linspace(-2,2,60)'; y3 = erf(x3); // la fct erreur
legende = "ellipse@cercle@fct erreur"
plot2d([x1 x2 x3],[y1 y2 y3],style=[1 2 3], frameflag=4, leg=legende)
xtitle("Encore des courbes ...","x","y")
```

Dans certains cas *frameflag* doit être accompagné du paramètre *rect*. Par exemple si vous voulez définir vous-même le rectangle de visualisation (au lieu de laisser `plot2d` s'en charger), il faut utiliser $rect = [x_{min}, y_{min}, x_{max}, y_{max}]$ en conjonction avec $frameflag = 1$ comme dans l'exemple suivant :

```
x = linspace(-5,5,200)';
y = 1 ./(1 + x.^2);
xbasc()
plot2d(x, y, frameflag=1, rect=[-6,0,6,1.1])
xtitle("La fonction de Runge")
```

Finalement voici toutes les valeurs possibles pour ce paramètre :

`frameflag=0`	on utilise l'échelle précédente (ou par défaut)
`frameflag=1`	échelle donnée par rect
`frameflag=2`	échelle calculée via les les max et min de `Mx` et `My`
`frameflag=3`	échelle isométrique calculée en fonction de rect
`frameflag=4`	échelle isométrique calculée via les max et min de `Mx` et `My`
`frameflag=5`	idem à 1 mais avec adaptation eventuelle pour graduation
`frameflag=6`	idem à 2 mais avec adaptation eventuelle pour la graduation
`frameflag=7`	idem à 1 mais les courbes précédentes sont redessinées
`frameflag=8`	idem à 2 mais les courbes précédentes sont redessinées

rmq : dans les cas 4 et 5, il y a une modification (éventuelle) du rectangle de visualisation de sorte que les graduations (qui sont toujours « calées » sur les extrémités) soient plus harmonieuses (c-a-d « tombent » sur des nombres qui s'expriment avec peu de décimales).

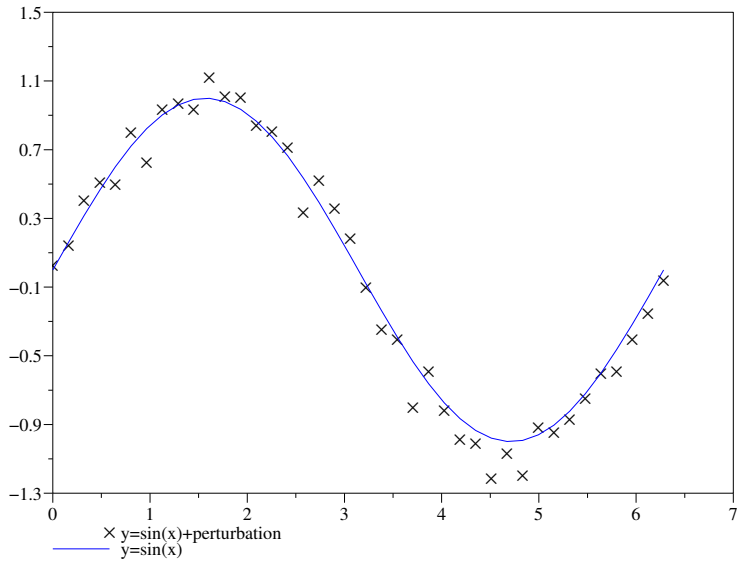

FIG. 4.3 – dessin en trait plain et avec des symboles non reliés

4. **préciser le placement des axes** : cela s'obtient avec la combinaison *axesflag=val*; dans l'exemple suivant (cf figure (4.5)) j'ai choisi val=5 qui permet d'obtenir des axes se croisant en $(0,0)$ [8] sans boite englobante :

```
x = linspace(-14,14,300)';
y = sinc(x);
xbasc()
plot2d(x, y, style=2, axesflag=5)
xtitle("La fonction sinc")
```

Voici le tableau donnant les diverses possibilités :

`axesflag=0`	pas de boite, ni d'axes et de graduations
`axesflag=1`	avec boite, axes et graduations (les x en bas, les y à gauche)
`axesflag=2`	avec boite mais sans axes ni graduations
`axesflag=3`	avec boite, axes et graduations (les x en bas, les y à droite)
`axesflag=4`	sans boite mais avec axes et graduations (tracés vers le milieu)
`axesflag=5`	sans boite mais avec axes et graduations (tracés en $y=0$ et $x=0$)

5. **utiliser une échelle logarithmique** : la séquence correspondante est *logflag=str* où str est une chaîne de deux caractères chacun pouvant prendre la valeur "n" (non log) ou "l" (log), le premier caractère règle le comportement sur l'axe des x, le deuxième celui de l'axe des y. Voici un exemple :

```
x = logspace(0,4,200)';
y = 1 ./x;
xbasc()
subplot(1,2,1)
   plot2d(x, y, style=2, logflag= "ln")
   xtitle("logflag=""ln""")
subplot(1,2,2)
   plot2d(x, y, style=2, logflag= "ll")
   xtitle("logflag=""ll""")
```

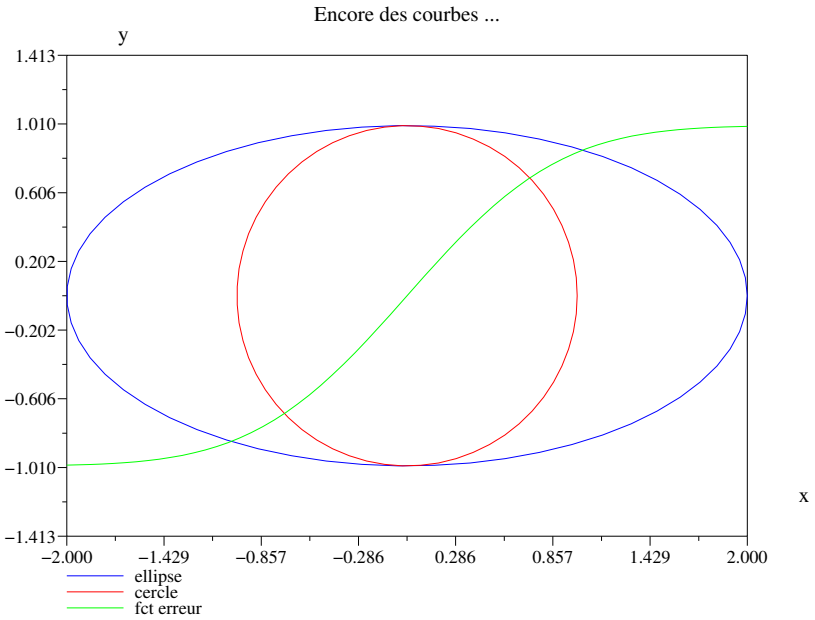

FIG. 4.4 – Ellipse, cercle et erf

Cet exemple vous montre aussi comment dessiner plusieurs graphes dans la même fenètre graphique en utilisant l'instruction *subplot(m, n, num)*. Le paramètre m correspond à la découpe verticale (en m parts égales), n à la découpe horizontale, et num au numéro de la sous-fenètre sélectionnée parmi les $m \times n$, les sous-fenètres étant numérotées de la gauche vers la droite puis de haut en bas (ainsi la sous-fenètre en position (i, j) a le numéro[9] $n \times (i-1) + j$). En fait rien n'interdit de modifier la grille au fur et à mesure des *subplot* pour obtenir ce que l'on cherche. Par exemple, avec :

```
xbasc()
subplot(1,2,1)
   titlepage("à gauche")
subplot(3,2,2)
   titlepage(["à droite";"en haut"])
subplot(3,2,4)
   titlepage(["à droite";"au centre"])
subplot(3,2,6)
   titlepage(["à droite";"en bas"])
xselect()
```

on scinde la fenètre verticalement en deux parties (gauche/droite), la sous-fenètre de droite étant découpée horizontalement en trois parts. Il faut en fait comprendre `subplot` comme une directive qui permet de sélectionner une portion de la fenètre graphique.

6. **le mot clé strf** : il permet de remplacer à la fois `frameflag` et `axesflag`, et, pour des raisons de compatibilité avec l'ancienne façon d'utiliser `plot2d`, il contient aussi un drapeau (flag) pour l'utilisation de la légende ou non. La valeur à donner est une chaîne de trois caractères `"xyz"` avec :

   x égal à 0 (pas de légende) ou 1 (légende à fournir avec la combinaison *leg=val*) ;

   y entre 0 et 9, correspond à la valeur à donner pour *frameflag* ;

   z entre 0 et 5, correspond à la valeur à donner pour *axesflag*.

   En fait il faut savoir l'utiliser car les séquences optionnelles de nombreuses primitives de dessin ne disposent pas encore des mots clés *frameflag* et *axesflag*. De plus il reste très pratique lorsque vous voulez rajouter un dessin sur un autre sans modifier l'échelle et le cadre ce qui s'obtient avec `strf="000"` (et évite donc d'écrire `frameflag=0, axesflag=0`).

[8] si le point $(0, 0)$ est dans le rectangle de visualisation !

[9] et non pas $m \times (i-1) + j$ comme indiqué dans la page d'aide !

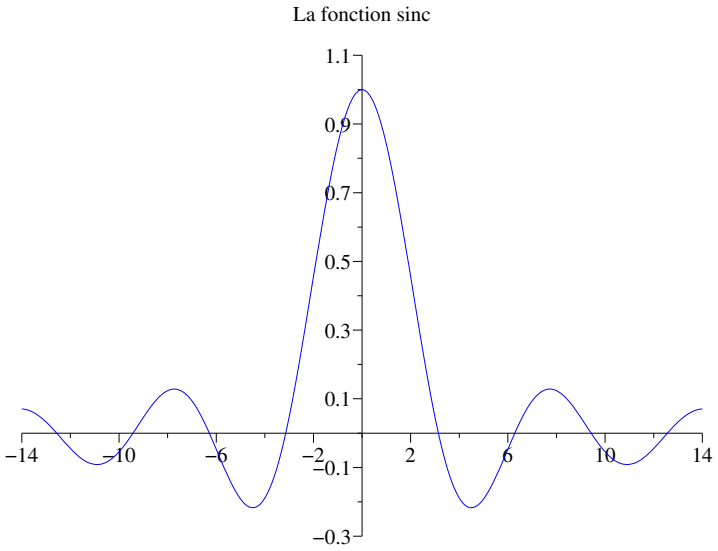

FIG. 4.5 – placement des axes obtenu avec axesflag=5

4.4 Des variantes de plot2d : plot2d2, plot2d3

Elles s'utilisent exactement comme `plot2d` : même syntaxe avec les mêmes arguments optionnels.

1. **plot2d2** permet de dessiner des fonctions en escalier : au lieu de tracer un segment de droite entre les points (x_i, y_i) et (x_{i+1}, y_{i+1}), `plot2d2` trace un segment horizontal (entre (x_i, y_i) et (x_{i+1}, y_i)) puis un segment vertical (entre (x_{i+1}, y_i) et (x_{i+1}, y_{i+1})). Voici un exemple (cf figure (4.7)) :

```
n = 10;
x = (0:n)';
y = x;
xbasc()
plot2d2(x,y, style=2, frameflag=5, rect=[0,-1,n+1,n+1])
xtitle("plot2d2")
```

2. **plot2d3** dessine des diagrammes en bâtons : pour chaque point (x_i, y_i) `plot2d3` trace un segment vertival entre $(x_i, 0)$ et (x_i, y_i) ; voici un exemple (cf figure (4.8)) :

```
n = 6;
x = (0:n)';
y = binomial(0.5,n)';
xbasc()
plot2d3(x,y, style=2, frameflag=5, rect=[-1,0,n+1,0.32])
xtitle("Probabilités pour la loi binomiale B(6,1/2)")
```

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

Avec `plot2d` et ses variantes, on ne peut pas dessiner en une seule fois plusieurs courbes qui n'ont pas été discrétisées avec le même nombre d'intervalles et l'on est obligé d'utiliser plusieurs appels successifs. Depuis la version 2-6, on peut se passer de spécifier l'échelle sans mauvaise surprise puisque, par défaut (*frameflag=8*) les courbes précédentes sont redessinées en cas de changement d'échelle. Cependant si on veut maîtriser l'échelle, il faut la fixer lors du premier appel puis utiliser `frameflag=0` pour les appels suivants[10]. Voici un exemple (cf figure (4.9)) :

[10] cette méthode est obligatoire si on veut une échelle iso-métrique.

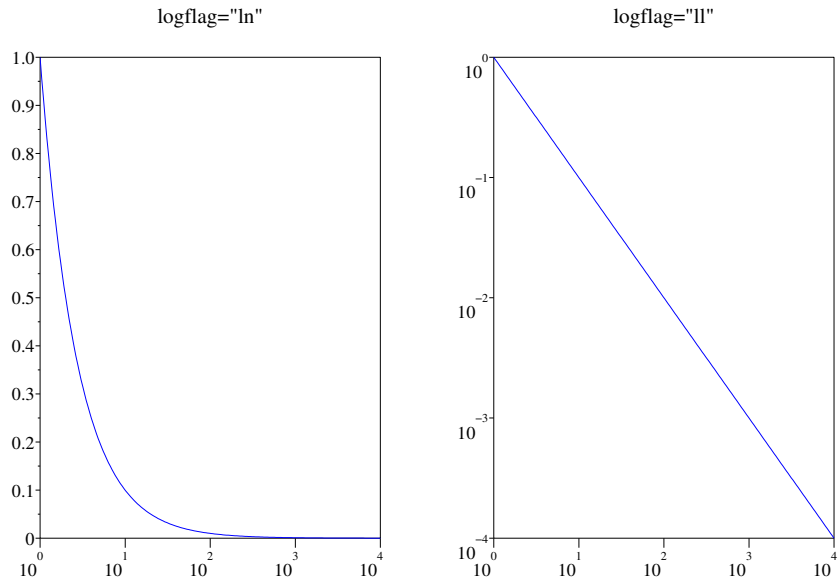

FIG. 4.6 – illustration pour le paramètre logflag

```
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);
y3 = x3.*(1-x3) + 0.1*(rand(x3)-0.5);  // idem a y2 avec une perturbation
ymin = min([y1 ; y2 ; y3]); ymax = max([y1 ; y2 ; y3]);
dy = (ymax - ymin)*0.05;          // pour se donner une marge
rect = [0,ymin - dy,1,ymax+dy];   // fenetre de visualisation
xbasc()                           // effacement des graphiques precedents
plot2d(x1, y1, style=1, frameflag=5, rect=rect) // 1er appel qui impose l'échelle
plot2d(x2, y2, style=2, frameflag=0)            // 2eme et 3 eme appel :
plot2d(x3, y3, style=-1,frameflag=0)            // on utilise l'echelle precedente
xtitle("Des courbes...","x","y")
```

Rmq : essayer cet exemple avec frameflag=1 au lieu de 5.

On ne peut pas alors avoir une légende pour chacune des courbes, mais cela reste possible en programmant un peu.

4.6 Jouer avec le contexte graphique

Si vous avez essayé les exemples précédents vous avez sans doute eu envie de modifier certaines choses comme la taille des symboles, la taille ou le type de la fonte utilisée pour le titre ou l'épaisseur des traits.

1. **Les fontes :** pour changer la fonte vous devez utiliser :

   ```
   xset("font",font_id,fontsize_id)
   ```

   où `font_id` et `fontsize_id` sont des entiers correspondants respectivement à la fonte et la taille choisies. La fonte courante s'obtient via :

   ```
   f=xget("font")
   ```

   où `f` est un vecteur, `f(1)` contenant l'identificateur de la fonte, et `f(2)` l'identificateur de la taille. On peut simplement changer / récupérer la taille avec `xset("font size",size_id)` et `fontsize_id = xget("font size")`.

   Voici ce qui est actuellement disponible :

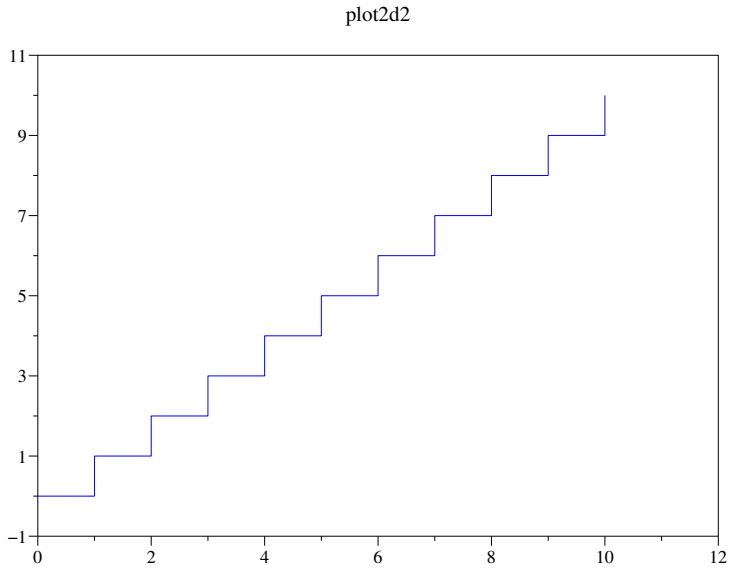

FIG. 4.7 – illustration pour plot2d2

nom de la fonte	Courier	Symbol	Times	Times-Italic	Times-Bold	Times-Bold-Italic
identificateur	0	1	2	3	4	5

taille	8 pts	10 pts	12 pts	14 pts	18 pts	24 pts
identificateur	0	1	2	3	4	5

Rmqs :
- la fonte Courier est à chasse fixe ;
- la fonte Symbol vous permet d'utiliser les lettres grecques (p correspondant à π, a à α, etc...) ;
- Times est la fonte par défaut et la taille par défaut est de 10 points.

2. **taille des symboles :** elle se change avec :

```
xset("mark size",marksize_id)
```

et on récupère la taille courante avec :

```
marksize_id = xget("mark size")
```

comme pour les tailles des fontes, les tailles des symboles s'échelonnent de 0 à 5, 0 étant la taille par défaut.

3. **épaisseur des traits :** elle se change / se récupère avec :

```
xset("thickness",thickness_id)
thickness_id = xget("thickness")
```

les tailles sont des entiers positifs, et correspondent au nombre de pixels utilisés pour l'épaisseur des tracés (1 par défaut). Un problème classique est que les tracés du cadre et des graduations réagissent aussi à ce paramètre alors que vous n'avez envie d'augmenter que l'épaisseur des courbes. Pour contourner ce problème on peut dessiner sans le cadre et les graduations (axesflag=0) puis revenir à l'épaisseur habituelle et faire un deuxième appel en gardant l'échelle précédente (frameflag=0) et dessiner une courbe qui n'apparaîtra pas dans le cadre fixé par le premier appel (par exemple une courbe réduite au seul point $(-\infty, -\infty)$) :

```
xset("thickness", 3)
plot2d(x, y, axesflag=0, ...)
xset("thickness", 1)
plot2d(-%inf, -%inf, frameflag=0, ...)
```

Le deuxième appel sert donc uniquement à tracer le cadre et les graduations.

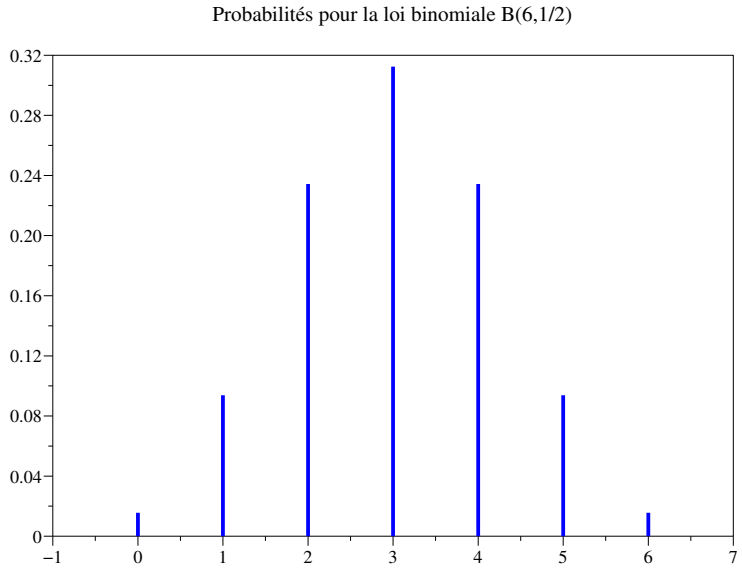

FIG. 4.8 – illustration pour plot2d3

4.7 Dessiner un histogramme

La fonction scilab adéquate s'appelle `histplot` et sa syntaxe est la suivante :

```
histplot(n, X, <,opt_arg>*)
```

où :

- `n` est soit un entier, soit un vecteur ligne (avec $n_i < n_{i+1}$) :
  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) ;
  2. dans le cas où `n` est un entier, les données sont comptabilisées dans les n classes équidistantes :

$$C_1 = [c_1, c_2],\ C_i =]c_i, c_{i+1}],\ i = 2, ..., n,\ \text{avec} \begin{cases} c_1 = min(X),\ c_{n+1} = max(X) \\ c_{i+1} = c_i + \Delta C \\ \Delta C = (c_{n+1} - c_1)/n \end{cases}$$

- `X` le vecteur (ligne ou colonne) des données à examiner ;
- `<,opt_arg>*` la séquence des arguments optionnels comme pour `plot2d` avec cependant la combinaison supplémentaire *normalization* = *val* où *val* est une constante (ou variable ou expression) booléenne (par défaut *vrai*). Lorsque l'histogramme est normalisé, son intégrale vaut 1 et approche donc une densité (dans le cas contraire, la valeur d'une plage correspond au nombre de composantes de X tombant dans cette plage). Plus précisemment, la plage de l'histogramme correspondant à l'intervalle C_i vaut donc (m étant le nombre de données, et $\Delta C_i = n_{i+1} - n_i$) :

$$\begin{cases} \frac{\text{card } \{X_j \in C_i\}}{m \Delta C_i} & \text{si } normalization = vrai \\ \text{card } \{X_j \in C_i\} & \text{si } normalization = faux \end{cases}$$

Voici un petit exemple, toujours avec la loi normale (cf figure (4.10)) :

```
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, style=2, frameflag=0, axesflag=0)
```

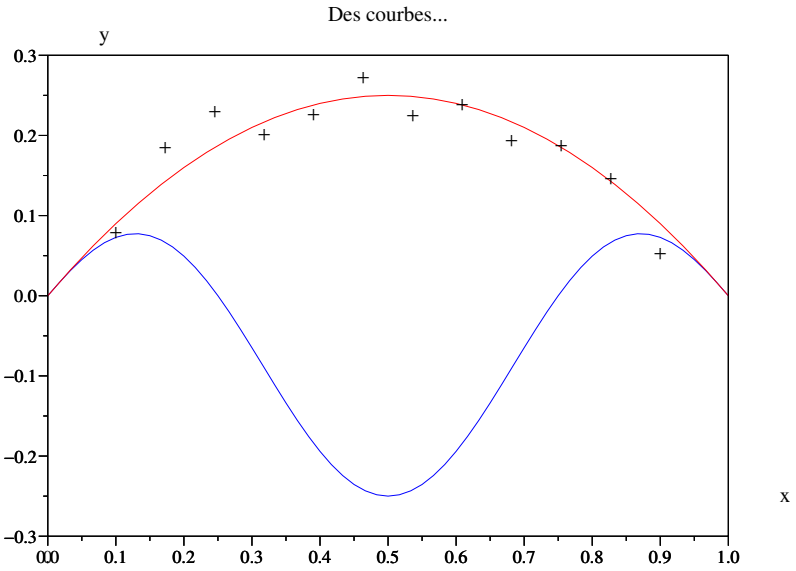

FIG. 4.9 – Encore des courbes...

4.8 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.

4.9 Animations simples

Il est facile de réaliser des petites animations avec Scilab qui permet l'utilisation de la technique du 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[11] :

- `Rec` qui enregistre toutes les opérations graphiques effectuées dans la fenêtre, et qui est le driver par défaut ;
- `X11` qui se contente simplement d'afficher les graphiques (il est alors imposible de « zoomer »).

Pour une animation il est souvent préférable d'utiliser ce dernier ce qui se fait par l'instruction `driver("X11")` (et `driver("Rec")` pour revenir au driver par défaut) .

Avec le double buffer, chacun des dessins successifs de l'animation se constitue d'abord dans une mémoire (appelée pixmap), puis, une fois terminé, la « pixmap » est basculée à l'écran. Voici un canevas simple pour une animation en Scilab :

```
driver("X11")       // pas d'enregistrement des opérations graphiques
xset("pixmap",1)    // on passe en mode double buffer
...........         // eventuellement une instruction pour fixer l'echelle
for i=1:nb_dessins
   xset("wwpc")     // effacement de la pixmap
   .......
   .......          // fabrication du (i eme) dessin
   .......
   xset("wshow")    // basculement de la pixmap à l'écran
end
xset("pixmap",0)    // retour a l'affichage direct à l'écran
driver("Rec")       // retour au driver par defaut
```

[11] en plus des drivers qui permettent de faire des dessins en postscript, en fig et en gif.

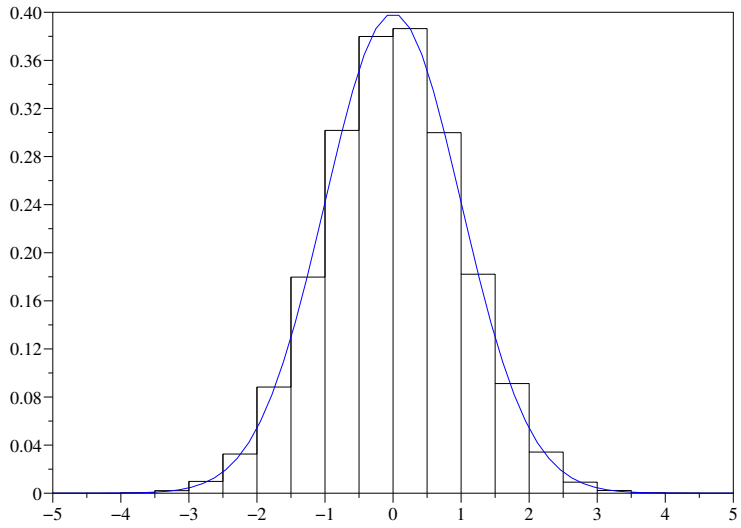

FIG. 4.10 – Histogramme d'un échantillon de nombres aléatoires suivants $N(0,1)$

Remarque : vous n'êtes pas obligé d'utiliser `xset("wwpc")` qui efface complètement le contenu de la pixmap ; par exemple vous pouvez juste effacer une partie rectangulaire avec `xclea` ce qui peut permettre d'éviter de reconstruire à chaque fois une partie statique du dessin. Pour utiliser la technique des masques, vous devez changer la fonction logique d'affichage avec `xset('alufunction',num)` (où `num` est un entier précisant la fonction à utiliser), voir le Help et les démos.

Voici un exemple d'animation : on fait se mouvoir le centre de gravité d'un rectangle (de longueur L et de largeur l) sur un cercle de rayon r et de centre $(0,0)$, le rectangle étant aussi animé d'un mouvement de rotation autour de son centre de gravité. Il y a certain un nombre de détails qui se greffent autour du canevas général exposé ci-avant :

- l'ordre `plot2d` sert uniquement à régler l'échelle (isométrique) pour les dessins ;
- `xset("background",1)` impose la couleur 1 (du noir dans la carte des couleurs par défaut) comme couleur d'arrière plan, mais il est recommandé d'exécuter l'instruction `xbasr()` pour mettre effectivement à jour la couleur du fond ;
- le dessin consiste à appeler la fonction `xfpoly` suivi de `xpoly` pour dessiner le bord (avec ici 3 pixels ce qui est obtenu avec `xset("thickness",3)`) ; à chaque fois on change la couleur à utiliser avec `xset("color",num)` ;
- l'instruction `xset("default")` repositionne le contexte graphique de la fenêtre à des valeurs par défaut ; ainsi la variable `pixmap` reprend la valeur 0, `thickness` la valeur 1, `background` sa valeur par défaut, etc...

```
n = 4000;
L = 0.6; l = 0.3; r = 0.7;
nb_tours = 4;
t = linspace(0,nb_tours*2*%pi,n)';
xg = r*cos(t); yg = r*sin(t);
xy = [-L/2  L/2 L/2 -L/2;...   // les 4 points du bord
      -l/2 -l/2 l/2  l/2];

xselect()
driver("X11")
xset("pixmap",1)
plot2d(%inf,%inf, frameflag=3, rect=[-1,-1,1,1], axesflag=0)
xset("background",1); // un fond noir
xbasr()               // utile pour la mise a jour du background
```

```
xset("thickness",3)   // augmentation de l'epaisseur des traits (3vpixels)

xset("font",2,4)
for i=1:n
   xset("wwpc")
   theta = 3*t(i);
   xyr = [cos(theta) -sin(theta);...
          sin(theta)  cos(theta)]*xy;
   xset("color",2)
   xfpoly(xyr(1,:)+xg(i), xyr(2,:)+yg(i))
   xset("color",5)
   xpoly(xyr(1,:)+xg(i), xyr(2,:)+yg(i),"lines",1)
   xset("color",32)
   xtitle("Animation simple")
   xset("wshow")   // basculement de la pixmap à l'écran
end
driver("Rec")      // retour au driver par defaut
xset("default")    // remet le contexte graphique par defaut
```

4.10 Les surfaces

L'instruction générique pour dessiner des surfaces est `plot3d`[12]. Avec une représentation de votre surface par facettes, on peut définir une couleur pour chaque facette. Depuis la version 2.6, on peut aussi, pour des facettes triangulaires ou quadrangulaires, spécifier une couleur par sommet et le rendu de la facette est obtenu par interpolation des couleurs définies aux sommets.

4.10.1 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
```

Vous devez alors obtenir quelque chose qui ressemble à la figure (4.11)[13]. De façon plus générale, on utilise :

```
plot3d(x,y,z <,opt_arg>*)
plot3d1(x,y,z <,opt_arg>*)
```

où, comme pour `plot2d`, `<,opt_arg>*` désigne la séquence des arguments optionnels, `opt_arg` prenant la forme *mot_clé=valeur*. Dans la forme la plus simple, `x` et `y` sont deux vecteurs lignes ($(1, nx)$ et $(1, ny)$) correspondants à la discrétisation en x et en y, et `z` est une matrice (nx, ny) telle que $z_{i,j}$ est « l'altitude » au point (x_i, y_j).

Voici les arguments optionnels possibles :

1. `theta=`*val_theta* et `alpha=`*val_alpha* sont les deux angles (en degré) précisant le point de vue en coordonnées sphériques (si O est le centre de la boite englobante, Oc la direction de la caméra, alors $\alpha = angle(Oz, Oc)$ et $\theta = angle(0x, Oc')$ où Oc' est la projection de Oc sur le plan Oxy ;
2. `leg=`*val_leg* permet d'écrire un label pour chacun des axes (exemple `leg="x@y@z"`), l'argument effectif *val_leg* est une chaîne de caractères où `@` est utilisé comme séparateur de labels ;
3. `flag=`*val_flag* où *val_flag* est un vecteur à trois composantes *[mode type box]* permet de préciser plusieurs choses :

[12] `plot3d1` qui s'utilise de façon quasi identique permet de rajouter des couleurs selon la valeur en z.

[13] 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

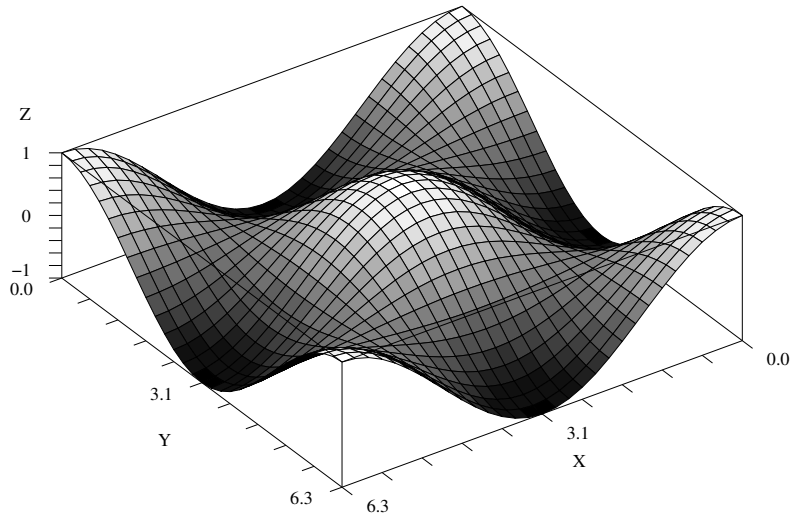

FIG. 4.11 – la fonction $z = cos(x)cos(y)$

(a) le paramètre *mode* est relatif au dessin des faces et du maillage :

i. avec $mode > 0$, les faces cachées sont « enlevées[14] », le maillage est visible ;

ii. avec $mode = 0$, on obtient un rendu « fil de fer » (wireframe) de la surface ;

iii. avec $mode < 0$, les faces cachées sont enlevées et le maillage n'est pas dessiné.

De plus le côté positif d'une face (voir plus loin) sera peint avec la couleur numéro *mode* alors que le côté opposé est peint avec une couleur que l'on peut changer avec l'instruction `xset("hidden3d",colorid)` (par défaut la couleur 4 de la carte).

(b) le paramètre *type* permet de définir l'échelle :

type	échelle obtenue
0	on utilise l'échelle précédente (ou par défaut)
1	échelle fournie avec `ebox`
2	échelle obtenue à partir des minima et maxima des données
3	comme avec 1 mais l'échelle est isométrique
4	comme avec 2 mais l'échelle est isométrique
5	variante de 3
6	variante de 4

(c) et enfin le paramètre *box* contrôle le pourtour du graphe :

box	effet obtenu
0	juste le dessin de la surface
2	des axes sous la surface sont dessinés
3	comme pour 2 avec en plus le dessin de la boite englobante
4	comme pour 3 avec en plus la graduation des axes

4. `ebox=`*val_ebox* permet de définir la boite englobante, *val_ebox* devant être un vecteur à 6 composantes $[x_{min}, x_{max}, y_{min}, y_{max}, z_{min}, z_{max}]$.

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`. Dans ce script j'utilise `flag=[2 4 4]`, c'est à dire avec :

[14] actuellement c'est l'algorithme du peintre qui est utilisé, c-a-d qu'un tri des facettes est effectué et les plus éloignées de l'observateur sont dessinées en premier.

– *mode* = 2 la surface sera peinte (côté positif) avec la couleur 2 et le maillage sera apparent ;
– *type* = 4 on utilise une échelle isométrique calculée via les données (ce qui doit être équivalent à choisir *type* = 3 avec un paramètre *ebox* obtenu en utilisant les minima et maxima des données) ;
– *box* = 4 le dessin apparaîtra avec une boite et des graduations.

```
x=linspace(-%pi,%pi,31);
z=sin(x)'*sin(x);
n = 200;
theta = linspace(30,390,n);  // un tour complet
alpha = [linspace(60,0,n/2) linspace(0,80,n/2)]; // vers le haut puis
                                                 // vers le bas
xselect()
xset("pixmap",1)   // pour activer le double buffer
driver("X11")
// on fait varier theta
for i=1:n
   xset("wwpc") // effacement du buffer courant
   plot3d(x,x,z,theta=theta(i),alpha=alpha(1),leg="x@y@z",flag=[2 4 4])
   xtitle("variation du point de vue avec le parametre theta")
   xset("wshow")
end
// on fait varier alpha
for i=1:n
   xset("wwpc") // effacement du buffer courant
   plot3d(x,x,z,theta=theta(n),alpha=alpha(i),leg="x@y@z",flag=[2 4 4])
   xtitle("variation du point de vue avec le parametre alpha")
   xset("wshow")
end
xset("pixmap",0)
driver("Rec")
```

4.10.2 La couleur

Vous pouvez réessayer les 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 ».

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 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. Enfin, deux fonctions, `hotcolormap` et `greycolormap` fournissent deux cartes avec une variation progressive des couleurs[15]. 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 de nouveau l'exemple 1

```
x = linspace(0,2*%pi,31);
z = cos(x)'*cos(x);
C = hotcolormap(32); // la hot colormap avec 32 couleurs
xset("colormap",C)
xset("hidden3d",30)  // choix de la couleur 30 pour les faces négatives
xbasc()
plot3d1(x,x,z, flag=[1 4 4])  // tester aussi avec flag=[-1 4 4]
```

Remarque : avec `plot3d1`, seul le signe du paramètre *mode* est utilisé (pour *mode* ≥ 0 le maillage apparaît et pour *mode* < 0 il n'est pas dessiné).

[15] voir aussi la section Contributions sur le site Scilab.

4.10.3 `plot3d` avec des facettes

Pour utiliser cette fonction 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, elle s'utilise comme précédemment pour les autres arguments :

```
plot3d(xf,yf,zf <,opt_arg>*)
```

Attention l'orientation des facettes est différente de la convention habituelle, cf figure (4.12).

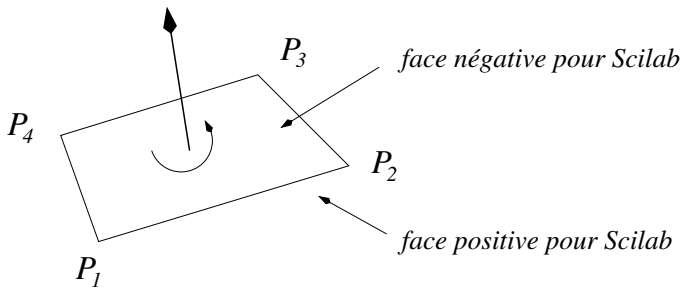

FIG. 4.12 – orientation des facettes en scilab

Pour définir une couleur 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 (dans la carte) de la couleur attribuée à la i$^{\text{ème}}$ facette.

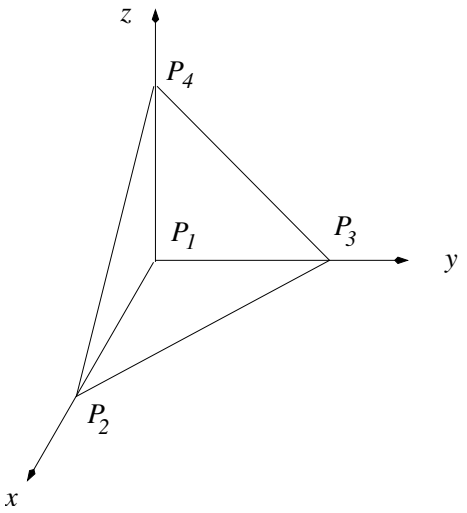

FIG. 4.13 – un tétraèdre

Comme premier exemple, visualisons les faces du tétraèdre de la figure (4.13), pour lequel :

$$P_1 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix},\ P_2 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix},\ P_3 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix},\ P_4 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix},$$

et définissons les faces comme suit (de sorte à obtenir les faces extérieures avec l'orientation positive pour Scilab) :

$$f_1 = (P_1, P_2, P_3),\ f_2 = (P_2, P_4, P_3),\ f_3 = (P_1, P_3, P_4),\ f_4 = (P_1, P_4, P_2)$$

On écrira alors :

```
//    f1 f2 f3 f4
xf = [ 0  1  0  0;
```

```
       1  0  0  0;
       0  0  0  1];
yf = [ 0  0  0  0;
       0  0  1  0;
       1  1  0  0];
zf = [ 0  0  0  0;
       0  1  0  1;
       0  0  1  0];  // ouf !

xbasc()
plot3d(xf,yf,list(zf,2:5), flag=[1 4 4], leg="x@y@z",alpha=30, theta=230)
xselect()
```

Avec ces paramètres vous devriez obtenir quelque chose qui ressemble à la figure 4.14. Vous pouvez remarquer que `plot3d` utilise une simple projection orthographique et non une projection perspective plus réaliste.

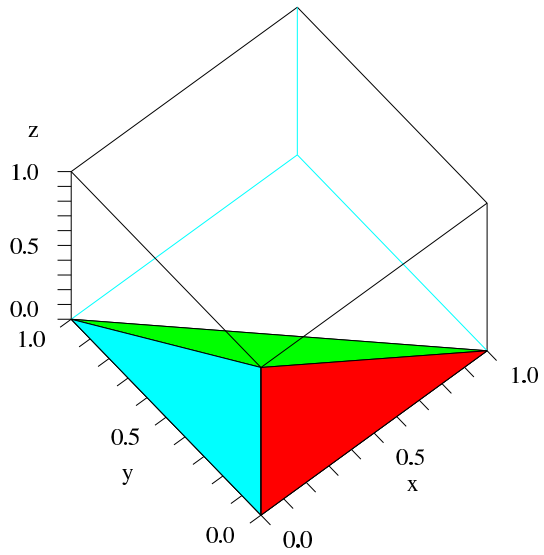

FIG. 4.14 – le tétraèdre dessiné avec Scilab

Pour des besoins courants, le calcul des facettes peut être effectués avec les fonctions suivantes :
– `eval3dp` et `nf3d` pour les surfaces définies par $x = x(u, v), y = y(u, v), z = z(u, v)$ (voir 4.10.4) ;
– `genfac3d` pour les surfaces définies par $z = f(x, y)$ (un exemple est proposé plus loin (4.10.5)).

Si votre surface (polyédrique) est définie comme mon cube de l'exemple sur les tlists, vous ne pouvez pas la visualiser directement avec `plot3d`. Pour obtenir la description attendue par Scilab, vous pouvez utiliser une fonction comme celle-ci :

```
function [xf,yf,zf] = facettes_polyedre(P)
   // transforme la structure Polyedre de l'exemple
   // sur les tlist en description de facettes pour
   // visualisation avec plot3d
   [ns, nf] = size(P.face)  // ns : nb de sommets par facette
                            // nf : nb de facettes
   xf=zeros(P.face); yf=zeros(P.face); zf=zeros(P.face)
   for j=1:ns
      num =  connect(ns+1-j,:) // pour inverser l'orientation
      xf(j,:) = P.coord(1, num)
      yf(j,:) = P.coord(2, num)
      zf(j,:) = P.coord(3, num)
   end
```

```
endfunction
```

En définissant le cube comme précédemment, vous obtiendrez alors son dessin avec :

```
[xf,yf,zf] = facettes_polyedre(Cube);
plot3d(xf,yf,list(zf,2:7), flag=[1 4 0],theta=50,alpha=60)
```

4.10.4 Dessiner une surface définie par $x = x(u, v)$, $y = y(u, v)$, $z = z(u, v)$

Réponse : prendre une discrétisation du domaine des paramètres et calculer les facettes avec la fonction (`eval3dp`). Pour des raisons d'efficacité, la fonction qui définit le paramètrage de votre surface doit être écrite « vectoriellement ». Si $(u_1, u_2, \ldots, u_m)$ et $(v_1, v_2, \ldots, v_n)$ 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 $m \times n$:

$$U = (\underbrace{u_1, u_2, \ldots, u_m}_{1}, \underbrace{u_1, u_2, \ldots, u_m}_{2}, \ldots\ldots, \underbrace{u_1, u_2, \ldots, u_m}_{n})$$
$$V = (\underbrace{v_1, v_1, \ldots, v_1}_{m \text{ fois } v_1}, \underbrace{v_2, v_2, \ldots, v_2}_{m \text{ fois } v_2}, \ \ldots\ \ldots\ , \underbrace{v_n, v_n, \ldots, v_n}_{m \text{ fois } v_n})$$

A partir de ces deux vecteurs, votre fonction doit renvoyer 3 vecteurs X, Y et Z de longueur $m \times n$ tels que :

$$X_k = x(U_k, V_k), Y_k = y(U_k, V_k), Z_k = z(U_k, V_k)$$

Voici quelques exemples de paramétrisation de surfaces, écrite[16] de façon à pouvoir être utilisée avec `eval3dp` :

```
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)
endfunction

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
endfunction

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)
endfunction

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)
```

[16] en fait vous pouvez écrire ces équations naturellement puis remplacer les `*` et `/` par des `.*` et `./` et ça marchera !

```
  y = (R + r.*cos(phi)).*sin(theta)
  z = r.*sin(phi)
endfunction
```

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()
plot3d1(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.

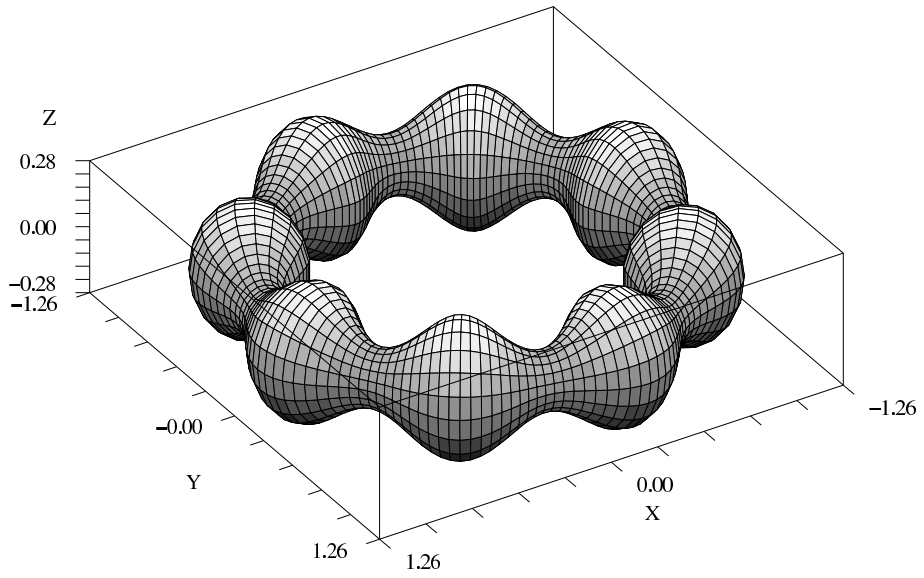

FIG. 4.15 – Un tore bosselé...

La fonction `nf3d` est un peu analogue à `eval3dp`, mais, à partir d'une discrétisation de u et v il faut définir soit-même des matrices X, Y, Z telles que :

$$\begin{aligned} X_{i,j} &= x(u_i, v_j) \\ Y_{i,j} &= y(u_i, v_j) \\ Z_{i,j} &= z(u_i, v_j) \end{aligned}$$

et vos facettes s'obtiennent alors avec `[xf,yf,zf] = nf3d(X,Y,Z)`. Comme exemple, voici le ruban de Moëbius définit juste avant :

```
nt = 120;
nr = 10;
rho = linspace(-0.5,0.5,nr);
theta = linspace(0,2*%pi,nt);
R = 1;
X = (R + rho'*sin(theta/2)).*(ones(nr,1)*cos(theta));
Y = (R + rho'*sin(theta/2)).*(ones(nr,1)*sin(theta));
Z = rho'*cos(theta/2);
[xf,yf,zf] = nf3d(X,Y,Z);
```

```
xbasc()
plot3d(xf,yf,zf, flag=[2 4 6], alpha=60, theta=50)
xselect()
```

Remarque : pour obtenir les bonnes matrices, j'ai été obligé d'utiliser la fonction `ones`, ce qui ne rend pas le code très clair : la fonction `eval3dp` est plus simple à utiliser !

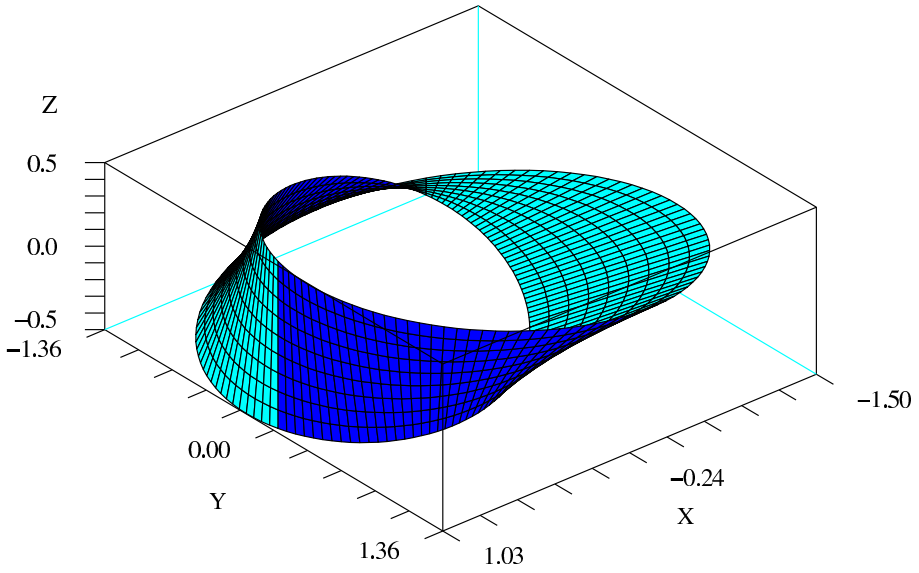

FIG. 4.16 – Le ruban de Moëbius

4.10.5 `plot3d` avec interpolation des couleurs

Depuis la version 2.6, il est maintenant possible d'associer une couleur pour chaque sommet d'une facette. Pour cela il suffit de donner une matrice `colors` de même taille que les matrices `xf, yf, zf` donnant la description par facette, c-a-d telle que `colors(i,j)` soit la couleur associée au $i^{\text{ème}}$ sommet de la $j^{\text{ème}}$ face, et de l'associer au troisième argument (`zf`) avec une liste :

```
plot3d(xf,yf,list(zf,colors) <,opt_arg>*)
```

Voici l'exemple initial de `plot3d` avec affichage sans le maillage et avec :
- une couleur par face pour le dessin de gauche,
- une couleur par sommet pour celui de droite.

Pour calculer les couleurs, j'utilise une petite fonction qui me permet d'associer linéairement des valeurs à la carte graphique courante (j'utilise la fonction `dsearch` disponible depuis la version 2.7 mais vous pouvez facilement vous en passer). Vous noterez aussi l'utilisation de la fonction `genfac3d` qui permet de calculer les facettes.

```
// exemple pour illustration de plot3d avec interpolation de couleur
function [col] = associe_couleur(val)
   // associe une couleur pour chaque valeur de val
   n1 = 1                    // numero de la 1 ere couleur
   n2 =  xget("lastpattern") // numéro de la derniere couleur
   nb_col = n2 - n1 + 1
   classes = linspace(min(val),max(val),nb_col)
   col = dsearch(val, classes)
endfunction

x=linspace(0,2*%pi,31);
z=cos(x)'*cos(x);
[xf,yf,zf] = genfac3d(x,x,z);
xset("colormap",graycolormap(64))
```

```
zmeanf = mean(zf,"r");
zcolf = associe_couleur(zmeanf);
zcols = associe_couleur(zf);

xbasc()
xset("font",6,2)  // la fonte 6 (helvetica) n'est disponible
                  // que dans la version cvs de scilab
subplot(1,2,1)
   plot3d(xf,yf,list(zf,zcolf), flag=[-1 4 4])
   xtitle("Une couleur par face")
subplot(1,2,2)
   plot3d(xf,yf,list(zf,zcols), flag=[-1 4 4])
   xtitle("Une couleur par sommet")
xselect()
```

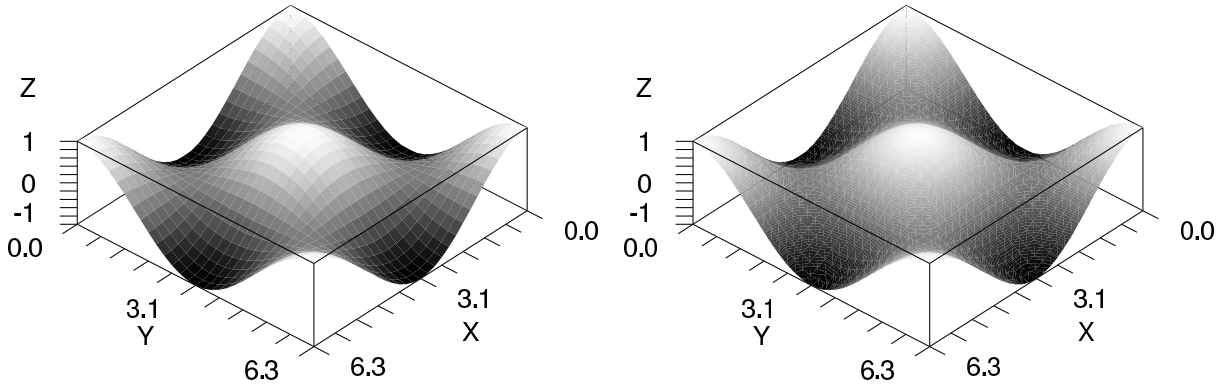

FIG. 4.17 – Avec et sans interpolation des couleurs

4.11 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 <,opt_arg>*)
param3d1(x,y,list(z,colors) <,opt_arg>*)
```

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`, modulo le fait que `flag` ne ne comporte pas de paramètre *mode*.

`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 $i^{ème}$ courbe est dessinée avec la $i^{ème}$ 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 (4.18) :

```
t = linspace(0,4*%pi,100)';
x1 = cos(t); y1 = sin(t) ; z1 = 0.1*t;  // une helice
x2 = x1 + 0.1*(1-rand(x1));
```

```
y2 = y1 + 0.1*(1-rand(y1));
z2 = z1 + 0.1*(1-rand(z1));
xbasc();
xset("font",2,3)
param3d1([x1 x2],[y1 y2],list([z1 z2], [1,-9]), flag=[4 4])
xset("font",4,4)
xtitle("Helice avec perles")
```

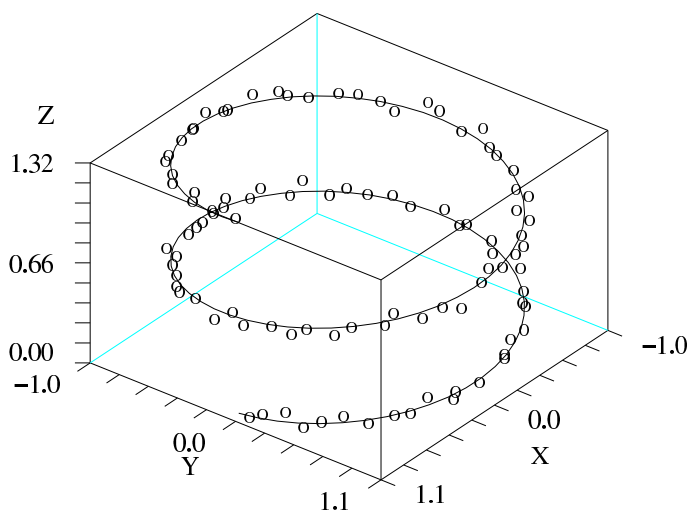

FIG. 4.18 – Courbe et points dans l'espace...

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 :

```
n = 50;        // nombre de points
P = rand(n,3); // des points au hasard
// on impose les dimensions de la boite englobante
ebox = [0 1 0 1 0 1];
// ici je separe les points en 2 groupes pour montrer comment mettre des
// symboles et des couleurs differentes pour les points
m = 30;
P1 = P(1:m,:) ; P2 = P(m+1:n,:);

// le dessin
xbasc()
// premier groupe de points
xset("color",2)  // du bleu avec la carte par defaut
param3d1(P1(:,1),P1(:,2),list(P1(:,3), -9), alpha=60, theta=30,...
         leg="x@y@z", flag=[3 4], ebox=ebox)
      // flag=[3 4] : 3 -> echelle iso se basant sur ebox
      //              4 -> boite + graduation
// pour le deuxieme groupe
xset("color",5)  // du rouge avec la carte par defaut
param3d1(P2(:,1),P2(:,2),list(P2(:,3), -5), flag=[0 0])
     // -5 pour des triangles inverses
     // [0 0]  : echelle fixée et cadre dessiné avec l'appel précédent
xset("color",1) // pour remettre le noir comme couleur courante
xtitle("Des points...")
xselect()
```

4.12 Divers

Il existe encore beaucoup de primitives graphiques dont :

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

Pour se rendre compte des possibilités[17] il suffit de parcourir la rubrique Graphic Library de l'aide. Depuis la version 2.7 il existe un nouveau mode graphique « orienté objet » qui permet de modifier les propriétés d'un graphique après l'avoir dessiné. Par défaut il n'est pas activé mais si vous voulez l'expérimenter il faut exécuter l'instruction :

```
set("figure_style","new")
```

avant de faire un dessin. Comme ce nouveau mode est en cours de développement il est préférable d'utiliser les versions cvs de scilab.

Enfin, la bibliothèque d'Enrico Ségré, que vous pouvez récupérer sur sa page personnelle :

```
http://www.weizmann.ac.il/~fesegre/
```

complète celle de Scilab et contient aussi des fonctions permettant de simplifier certaines taches.

[17] attention risque de noyade !

Chapitre 5

Applications et compléments

Ce chapitre se propose de vous montrer comment résoudre certains problèmes types d'analyse numérique avec Scilab (en fait uniquement des équations différentielles actuellement...) et apporte des compléments pour pouvoir écrire des petites simulations stochastiques.

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.
endfunction
```

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)
endfunction
```

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();
```

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

[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 »

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` en rajoutant un argument optionnel comme pour `plot2d` (il faut utiliser `strf=`*val_strf* car les paramètres *frameflag* et *axesflag* ne sont pas supportés). 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, strf="041")
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, style=-9, strf="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,:)', style=couleurs(num+1),  strf="000")
   elseif c_i == 2 then
      break
   end
end
```

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)
endfunction
```

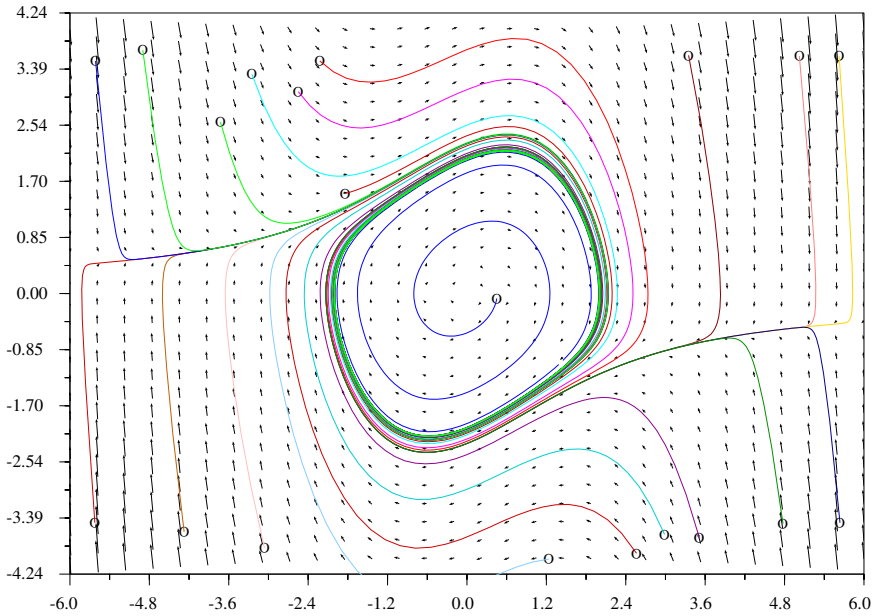

FIG. 5.1 – Quelques trajectoires dans le plan de phase pour l'équation de Van der Pol

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, $t_{k-1} \rightarrow t_k = t_{k-1} + \Delta t_k$, 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, strf="041")
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
   [c_i,c_x,c_y]=xclick();
   if c_i == 0 then
      plot2d(c_x, c_y, style=-9, strf="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,:)', style=couleurs(num+1), strf="000")
   elseif c_i == 2 then
      break
   end
end
```

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] :

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

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

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

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

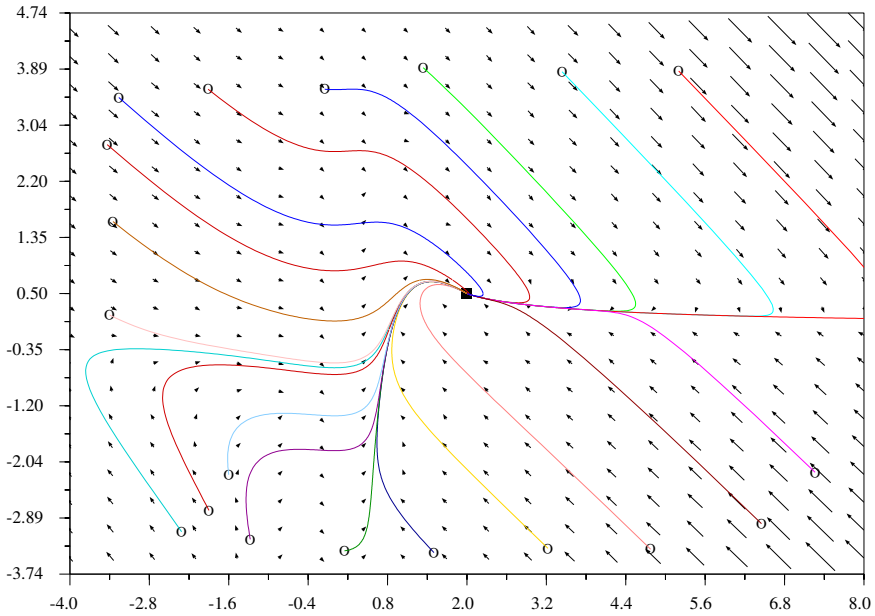

FIG. 5.2 – Quelques trajectoires dans le plan de phase pour le Brusselator ($\epsilon = -4$)

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 on peut par exemple additionner ces nombres entre-eux, ce qui donne :

```
v = getdate()
rand("seed", sum(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 ;
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"`.

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[5] 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)[7] :

```
function [X] = G(p)
   // loi geometrique
   X = 1
   while rand() > p   // echec
      X = X+1
   end
endfunction
```

[5] mais peu efficace pour N grand !

[6] d'une manière générale on utilisera plutôt la fonction `grand` permet d'obtenir la plupart des lois classiques.

[7] cette fonction est peu efficace pour p petit.

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")
```

5.2.2 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. Il est alors préférable d'utiliser `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 n réalisations, sous la forme d'un vecteur colonne) :

1. une loi uniforme sur les entiers d'un grand intervalle $[0, m[$:

   ```
   X = grand(n,1,"lgi")
   ```

   où m dépend du générateur de base (par défaut $m = 2^{32}$ (voir plus loin)) ;
2. une loi uniforme sur les entiers de l'intervalle $[k_1, k_2]$:

   ```
   X = grand(n,1,"uin",k1,k2)
   ```

   (il faut que $k_2 - k_1 \leq 2147483561$ mais dans le cas contraire ce problème est signalé par un message d'erreur) ;
3. pour la loi uniforme sur $[0, 1[$:

   ```
   X = grand(n,1,"def")
   ```
4. pour la loi uniforme sur $[a, b[$:

   ```
   X = grand(n,1,"unf",a,b)
   ```
5. pour la loi binomiale $B(N, p)$:

   ```
   X = grand(n,1,"bin",N,p)
   ```
6. pour la loi géométrique $G(p)$:

   ```
   X = grand(n,1,"geom",p)
   ```
7. pour la loi de Poisson de moyenne μ :

   ```
   X = grand(n,1,"poi",mu)
   ```
8. pour la loi exponentielle de moyenne λ :

   ```
   X = grand(n,1,"exp",lambda)
   ```
9. pour la loi normale de moyenne μ et d'écart type σ :

   ```
   X = grand(n,1,"nor",mu,sigma)
   ```

Il y en a d'autres (cf la page d'aide).

Depuis la version 2.7, `grand` est muni de différents générateurs de base (qui fournissent des entiers selon la loi lgi). Par défaut `grand` utilise *Mersenne Twister* qui possède une période gigantesque de 2^{19937} et il y a en tout 6 générateurs[8].

Pour opérer sur ces générateurs de base, vous pouvez utiliser les instructions suivantes :

[8] `"mt"` ,`"kiss"`, `"clcg4"`, `"clcg2"`, `"fsultra"`, et `"urand"`, ce dernier étant simplement le générateur de `rand`.

`nom_gen = grand("getgen")`	permet de récupérer le (nom du) générateur courant
`grand("setgen",nom_gen)`	le générateur `nom_gen` devient le générateur courant
`etat = grand("getsd")`	permet de récupérer l'état interne du générateur courant
`grand("setsd",e1,e2,..)`	impose l'état interne du générateur courant

La dimension de l'état interne de chaque générateur dépend du type du générateur : de un entier pour *urand* à 624 entiers plus un index pour *Mersenne Twister*[9]. Si vous voulez refaire exactement la même simulation il faut connaître l'état initial (avant la simulation) du générateur utilisé et le sauvegarder d'une façon ou d'une autre. Exemple :

```
grand("setgen","kiss")   // kiss devient le générateur courant
e = [1 2 3 4];           // etat que je vais imposer pour kiss
                         // (il lui faut 4 entiers)
grand("setsd",e(1),e(2),e(3),e(4));    // voila c'est fait !
grand("getsd")           // doit retourner le vecteur e
X = grand(10,1,"def");   // 10 nombres
s1 = sum(X);
X = grand(10,1,"def");   // encore 10 nombres
s2 = sum(X);
s1 == s2                 // en général s1 sera different de s2

grand("setsd",e(1),e(2),e(3),e(4));   // retour à l'état initial
X = grand(10,1,"def");   // de nouveau 10 nombres
s3 = sum(X);
s1 == s3                 // s1 doit etre egal a s3
```

5.3 Les fonctions de répartition 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.
Remarques :

- le fait de travailler à la fois avec p et $q = 1 - p$ permet d'obtenir de la précision dans les zones où p est proche de 0 ou de 1. Lorsque p est proche de 0 la fonction travaille en interne avec p mais avec lorsque p est proche de 1 la fonction travaille en interne avec q ;
- la chaîne de caractères permettant d'obtenir la fonction inverse n'est pas toujours `"X"`... voyez les pages d'aide correspondantes.

[9] une procédure d'initialisation avec un seul entier existe néanmoins pour ce générateur.

5.4 Simulations stochastiques simples

5.4.1 Introduction et notations

Souvent une simulation va consister en premier lieu, à obtenir un vecteur :

$$x^m = (x_1, ..., x_m)$$

dont les composantes sont considérées comme les réalisations de variables aléatoires indépendantes et de même loi $X_1, X_2, ..., X_m$ (on notera X une variable aléatoire générique suivant la même loi). Dans la pratique le vecteur x^m s'obtient directement ou indirectement à partir des fonctions `rand` ou `grand`[10].

A partir de l'échantillon x^m on cherche à approcher les caractéristiques de la loi sous-jacente comme l'espérance, l'écart type, la fonction de répartition (ou la densité) ou, si on emet une hypothèse sur la loi en question, à déterminer son ou ses paramètres, ou encore si les paramètres sont connus, à vérifier via un ou des tests statistiques que notre échantillon est (probablement) bien issu de v.a. qui suivent effectivement la loi en question, etc... Pour les cas qui nous intéressent (cas d'école) on connaît la plupart du temps les résultats théoriques exacts et la simulation sert en fait à illustrer un résultat, un théorème (lgn, TCL, etc...), le fonctionnement d'une méthode ou d'un algorithme, ...

5.4.2 Intervalles de confiance

Une fois l'espérance empirique obtenue à partir de notre échantillon (en scilab avec `x_bar_m = mean(xm)`), on aimerait connaître un intervalle Ic (souvent centré en $\bar{x}_m$) pour affirmer que :

$$E[X] \in I_c \text{ avec probabilité de } 1 - \alpha$$

où souvent $\alpha = 0.05$ ou 0.01 (intervalles de confiance à respectivement 95 % et 99 %). L'outil de base pour dériver de tels intervalles est le T.C.L.. Si on pose :

$$\bar{X}_m = \frac{1}{m} \sum_{i=1}^{m} X_i$$

la variable aléatoire moyenne (dont $\bar{x}_m$ est une réalisation), alors, la loi des grands nombres nous dit que $\bar{X}_m$ « converge » vers $E[X]$ et le T.C.L. (sous certaines conditions...), nous dit que :

$$\lim_{m \to +\infty} P(a < \frac{\sqrt{m}(\bar{X}_m - E[X])}{\sigma} \leq b) = \frac{1}{\sqrt{2\pi}} \int_a^b e^{-t^2/2} dt$$

(on a posé $Var[X] = \sigma^2$). Si m est suffisamment grand, on peut approcher cette probabilité en considérant la limite « atteinte[11] » :

$$P(a < \frac{\sqrt{m}(\bar{X}_m - E[X])}{\sigma} \leq b) \approx \frac{1}{\sqrt{2\pi}} \int_a^b e^{-t^2/2} dt$$

Si l'on cherche un intervalle de confiance "symétrique" :

$$\frac{1}{\sqrt{2\pi}} \int_{-a}^{a} e^{-t^2/2} dt = F_{N(0,1)}(a) - F_{N(0,1)}(-a) = 2F_{N(0,1)}(a) - 1 = 1 - \alpha$$

alors :

$$a_\alpha = F^{-1}_{N(0,1)}(1 - \frac{\alpha}{2}) \text{ ou encore } - F^{-1}_{N(0,1)}(\frac{\alpha}{2})$$

ce qui s'écrit en scilab :

[10] en fait en statistique l'échantillon est la plupart du temps obtenu à partir de mesures physiques (température, pression, ...), biométriques (tailles, poids), de sondages, etc... les données obtenues étant stockées dans des fichiers (ou dans des bases de données) ; certains logiciels (comme R) proposent de tels jeux de données (pour que les étudiants puissent se faire la main !) mais pas scilab à l'heure actuelle ; néanmoins les cas où on utilise de telles simulations sont quand même nombreux, par exemple pour étudier le comportement de certains systèmes à des entrées (ou des perturbations) aléatoires ou même pour résoudre des problèmes purement déterministes mais pour lesquels les méthodes d'analyse numérique sont trop compliquées ou impossible à mettre en oeuvre.

[11] pour certaines lois on a des critères d'application, par exemple si $X \sim Ber(p)$ alors l'approximation par la limite est suffisamment précise (pour ce que l'on cherche à en faire) à partir du moment où $\min(mp, m(1-p)) > 10$.

```
a_alpha = cdfnor("X", 0, 1, 1-alpha/2, alpha/2)
```

On obtient donc finalement[12] :

$$E[X] \in [\bar{x}_m - \frac{a_\alpha \sigma}{\sqrt{m}}, \bar{x}_m + \frac{a_\alpha \sigma}{\sqrt{m}}]$$

avec probabilité $1-\alpha$ environ (si l'approximation par la limite est correcte...).

Le problème est que l'on ne connaît généralement pas l'écart type... On utilise alors soit une majoration, ou encore l'estimation donnée par :

$$S_m = \sqrt{\frac{1}{m-1}\sum_{i=1}^{m}(X_i - \bar{X}_m)^2}$$

En remplaçant l'écart type σ par s_m (où s_m est la réalisation de S_m), on obtient alors un intervalle de confiance que l'on qualifie lui aussi d'empirique.

Il existe cependant des cas particuliers :

1. si les X_i suivent la loi normale $N(\mu, \sigma^2)$ alors :

$$\sqrt{m}\frac{\bar{X}_m - \mu}{S_m} \sim t(m-1)$$

où $t(k)$ est la loi de Student à k degrés de liberté. Dans ce cas, les diverses approximations précédentes n'existent plus (approximation par la limite et approximation de l'écart type) et on obtient alors :

$$\mu \in [\bar{x}_m - \frac{a_\alpha s_m}{\sqrt{m}}, \bar{x}_m + \frac{a_\alpha s_m}{\sqrt{m}}] \text{ avec probabilité } 1-\alpha$$

où s_m est l'écart type empirique de l'échantillon (`sm = st_deviation(xm)` en scilab) et où le a_α est calculé à partir de la loi de Student (au lieu de la loi $N(0,1)$) :

$$a_\alpha = F^{-1}_{t(m-1)}(1 - \frac{\alpha}{2})$$

ce qui s'obtient en scilab[13] par :

```
a_alpha = cdft("T", m-1, 1-alpha/2, alpha/2)
```

2. lorsque la variance s'exprime en fonction de l'espérance (Bernouilli, Poisson, exponentielle,...) on peut se passer de l'approximation de l'écart type et l'intervalle s'obtient alors en résolvant une inéquation.

5.4.3 Dessiner une fonction de répartition empirique

La fonction de répartition de X est la fonction :

$$F(x) = \text{Probabilité que } X \leq x$$

La fonction de répartition empirique définie à partir de l'échantillon x^m est définie par :

$$F_{x^m}(x) = card\{x_i <= x\}/m$$

C'est une fonction en escalier qui se calcule facilement si on trie le vecteur x^m dans l'ordre croissant (on a alors $F_{x^m}(x) = i/m$ pour $x_i \leq x < x_{i+1}$). L'algorithme standard de tri de Scilab est la fonction `sort` qui trie dans l'ordre décroissant[14]. Pour trier le vecteur x^m dans l'ordre croissant, on utilise :

```
xm = - sort(-xm)
```

[12] Pour un intervalle à 95%, on a $a_\alpha \simeq 1.96$ que l'on approche souvent par 2.

[13] voir la page d'aide correspondante : en fait les fonctions `cdf` n'ont pas une syntaxe très réguliere et `"X"` n'est pas toujours l'indicateur permettant d'obtenir la fonction de répartition inverse, ici c'est`"T"` !

[14] voir aussi la fonction `gsort` qui permet de faire plus de choses

La fonction `plot2d2` nous permet alors de dessiner cette fonction sans se fatiguer. Voici un code possible :

```
function repartition_empirique(xm)
   //  tracé de la fonction de repartition (empirique)
   //  associée à l'échantillon xm
   m = length(xm)
   xm = - sort(-xm(:))
   ym = (1:m)'/m
   plot2d2(xm, ym, leg="repartition empirique")
endfunction
```

comportant une petite astuce : le `xm( :)` permet de faire fonctionner le code même si on utilise la fonction à partir d'un vecteur ligne.

Voici maintenant un exemple avec la loi normale $\mathcal{N}(0,1)$:

```
m = 100;
xm = grand(m,1,"nor",0,1);
xbasc()
repartition_empirique(xm);  // dessin de la fct de repartition empirique
// les donnees pour tracer la fonction de repartition "exacte"
x = linspace(-4,4,100)';
y = cdfnor("PQ", x, zeros(x), ones(x));
plot2d(x, y, style=2)      // on rajoute la courbe sur le premier dessin
xtitle("Fonctions de répartition exacte et empirique")
```

qui m'a permis d'obtenir la figure (5.3).

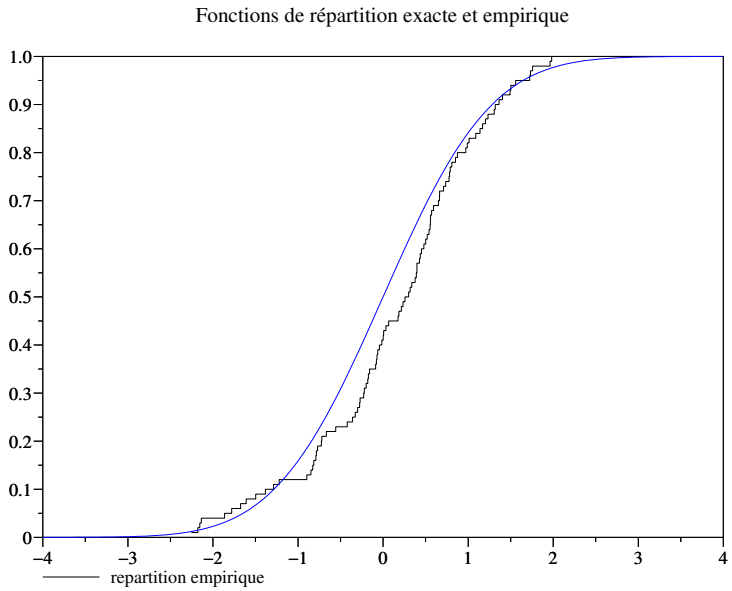

FIG. 5.3 – Fonctions de répartition exacte et empirique pour la loi normale

5.4.4 Test du χ^2

Soit donc $x^m = (x_1, ..., x_m)$ notre échantillon à analyser. Et soit l'hypothèse $\mathcal{H}$: « les variables aléatoires sous-jacentes $(X_1, ..., X_m)$ suivent la loi $\mathcal{L}$ ». On a envie de savoir si cette hypothèse est réaliste ou pas. Sur cet échantillon, on peut sans doute calculer les statistiques élémentaires (moyenne et écart type empirique) et si celles-ci semblent raisonnablement proches de l'espérance et de l'écart type de $\mathcal{L}$, on peut alors mettre en oeuvre un test statistique. Le test du χ^2 s'applique sur une loi discrète prenant

un nombre fini de valeurs. Par exemple supposons que la loi de $\mathcal{L}$ est donnée par $\{(v_i, p_i), 1 \leq i \leq n\}$. Le test consiste à calculer la quantité :

$$y = \frac{\sum_{i=1}^{n}(o_i - mp_i)^2}{mp_i}$$

où o_i est le nombre de résultats x_j égaux à v_i et à comparer la valeur obtenue y à un seuil y_α, le test étant alors positif[15] si $y \leq y_\alpha$.

Si on utilise, dans la formule donnant y, les variables aléatoires $X_1, ..., X_m$ au lieu de l'échantillon $x_1, ..., x_m$ on définit[16] alors une variable aléatoire Y qui suit approximativement (pour m suffisamment grand) la loi du χ^2 à $n-1$ degrés de liberté. La valeur seuil est alors obtenue par :

$$y_\alpha = F^{-1}_{\chi^2_{n-1}}(1-\alpha)$$

avec souvent $\alpha = 0.05$ où encore $\alpha = 0.01$. En Scilab, ce seuil s'obtiendra par :

```
y_seuil = cdfchi("X", n-1, 1-alpha, alpha)
```

Pour calculer les occurences o_i avec scilab, on pourra utiliser la méthode suivante :

```
occ = zeros(n,1);
for i=1:n
   occ(i) = sum(bool2s(xm == v(i))); // ou encore length(find(xm==v(i)))
end
if sum(occ) ~= m then, error("problème de comptage"), end
```

Et pour obtenir la quantité y on pourra écrire (en utilisant l'écriture vectorielle[17]) :

```
y = sum( (occ - m*p).^2 ./ (m*p) )
```

à la condition que `p` (le vecteur donnant les probabilités de $\mathcal{L}$), soit de même format que `occ` (ici un vecteur colonne vu mon choix pour `occ`).

Remarques :

- l'approximation par la loi du χ^2 est valable si m est suffisamment grand... on donne souvent la règle $mp_{min} > 5$ ($p_{min} = \min_i p_i$) comme condition d'application minimale de ce test ; ainsi vous pouvez vérifier cette condition et écrire un message pour prévenir l'utilisateur si elle n'est pas satisfaite.
- il est facile de regrouper ces calculs dans une fonction ;
- pour une loi continue le test peut s'appliquer en regroupant les valeurs par intervalles, par exemple pour la loi $U_{[0,1]}$, on utilise n intervalles équidistants ; de même pour une loi discrète prenant un nombre infini de valeurs, on peut regrouper celles en queue de la distribution ; ce même procédé peut s'appliquer aux lois discrètes finies pour lesquelles la condition d'application du test n'est pas satisfaite.
- si vous faites le test à partir d'une loi dont certains paramètres ont été déterminés avec les données, il faut retrancher d'autant le nombre de degrés de liberté pour le χ^2 ; par exemple si la loi attendue est $B(n-1, p)$ et que vous utilisiez $p = \bar{x}_m/(n-1)$ alors le seuil à ne pas dépasser pour un test positif serait de $y_\alpha = F^{-1}_{\chi^2_{n-2}}(1-\alpha)$ au lieu de $y_\alpha = F^{-1}_{\chi^2_{n-1}}(1-\alpha)$.

5.4.5 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 copies indépendantes de X. A partir des réalisations des X_i (disons le vecteur $x^m = (x_1, ..., x_m)$), on peut construire une fonction de répartition empirique qui doit converger lorsque ($m \to +\infty$) vers la fonction

[15] d'un point de vue intuitif, si l'hypothèse est bonne on s'attend à ce que o_i ne soit pas trop loin de mp_i, donc si l'hypothèse est fausse, on s'attend à obtenir une valeur élevée pour y, d'où le rejet de l'hypothèse pour $y > y_\alpha$...

[16] via la variable aléatoire vectorielle $O = (O_1, ..., O_n)$ qui suit une loi multinomiale.

[17] exercice : décomposer cette instruction vectorielle pour comprendre comment (et pourquoi) elle fonctionne.

de répartition exacte. Le test KS consiste justement à mesurer un écart entre la fonction de répartition exacte et la fonction de répartition empirique (obtenue avec notre échantillon $(x_1, ..., x_m)$) défini par :

$$k_m = \sqrt{m} \sup_{-\infty<x<+\infty} |F(x) - F_{x^m}(x)|$$

et à le comparer à une valeur « admissible ». Si on remplace nos réalisations par les variables aléatoires correspondantes, il est clair que k_m est en fait une variable aléatoire (que l'on notera K_m). La théorie nous dit qu'à la limite, sa loi a la fonction de répartition suivante :

$$\lim_{m\to+\infty} P(K_m \leq x) = H(x) = 1 - 2\sum_{j=1}^{+\infty}(-1)^{j-1}e^{-2j^2x^2}$$

Comme pour le test du χ^2, si l'hypothèse envisagée est fausse, la valeur obtenue pour k_m aura tendance à être grande, et on va alors rejetter l'hypothèse lorsque :

$$k_m > H^{-1}(1-\alpha)$$

avec $\alpha = 0.05$ ou 0.01 par exemple. Si on utilise l'approximation $H(x) \simeq 1 - 2e^{-2x^2}$ alors la valeur seuil à ne pas dépasser est :

$$k_{seuil} = \sqrt{\frac{1}{2}\ln(\frac{2}{\alpha})}$$

Le calcul de k_m ne pose pas de problème si on trie le vecteur $(x_1,\ x_2,\ \ldots,\ x_m)$. Supposons ce tri effectué, en remarquant que :

$$\sup_{x\in[x_i,x_{i+1}[} F_{x^m}(x) - F(x) = \frac{i}{m} - F(x_i),\ \text{et} \sup_{x\in[x_i,x_{i+1}[} F(x) - F_{x^m}(x) = F(x_{i+1}) - \frac{i}{m}$$

les deux quantités suivantes se calculent facilement :

$$k_m^+ = \sqrt{m} \sup_{-\infty<x<+\infty} (F_{x^m}(x) - F(x)) = \sqrt{m} \max_{1\leq j\leq m} (\frac{j}{m} - F(x_j))$$

$$k_m^- = \sqrt{m} \sup_{-\infty<x<+\infty} (F(x) - F_{x^m}(x)) = \sqrt{m} \max_{1\leq j\leq m} (F(x_j) - \frac{j-1}{m})$$

et l'on obtient alors $k_m = \max(k_m^+,\ k_m^-)$.

5.4.6 Exercices

Dé truqué ou non ?

On a effectué 200 fois l'expérience suivante avec le même dé : on le jette autant de fois qu'il le faut jusqu'à obtenir un 1 (mais on arrête lorsque le 1 n'est pas sorti au bout de 10 lancés). On a obtenu les résultats suivants :

nombre de jets	1	2	3	4	5	6	7	8	9	10	≥ 11
nombre d'expériences	36	25	26	27	12	12	8	7	8	9	30

par exemple il y a 36 expériences où le 1 est sorti lors du premier lancé, 25 où le 1 est sorti au deuxième lancé, etc...

Effectuer un test χ^2 pour essayer de répondre à la question.

Urne de Polya

On effectue 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 :

1. $E(X_N) = E(X_0) = X_0 = 1/2$;
2. X_N suit une loi uniforme sur $\{\frac{1}{N+2}, ..., \frac{N+1}{N+2}\}$;
3. pour $N \to +\infty$, X_N converge p.s. vers la loi uniforme sur $[0, 1)$.

Vous allez mettre en oeuvre une simulation pour illustrer les deux premiers résultats.

1. 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
endfunction
```

   mais pour effectuer des statistiques conséquentes cette fonction va être appelée trés souvent et comme les itérations sont lentes en Scilab, vous allez 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). On pourra utiliser la fonction `find` pour repérer les urnes où l'on a tiré une boule verte.
2. Ecrire un script pour retrouver par simulation le résultat attendu pour l'espérance (avec son intervalle de sécurité).
3. Continuer votre script en testant l'hypothèse H sur le comportement de la variable aléatoire X_N « H : X_N suit une loi uniforme sur $\{\frac{1}{N+2}, ..., \frac{N+1}{N+2}\}$ » avec un test du χ^2.
4. Essayer de réaliser des illustrations graphiques, par exemple, tracer les probabilités empiriques obtenues et les probabilités exactes sur un même dessin et, sur un autre, tracer la densité χ^2 tout en positionnant la valeur obtenue par le test ainsi que la valeur seuil par des traits verticaux.

Le pont brownien

Le processus stochatisque suivant (où U désigne la loi uniforme sur $[0, 1]$) :

$$X_n(t) = \frac{1}{\sqrt{n}} \sum_{i=1}^{n} (1_{\{U_i \leq t\}} - t)$$

est tel que pour $t \in]0, 1[$ fixé, on a :

$$\lim_{n \to +\infty} X_n(t) = Y(t) \sim \mathcal{N}(0, t(1-t))$$

On cherche à illustrer ce résultat à l'aide de simulations.

Travail à réaliser :

1. Ecrire une fonction Scilab `function [X] = pont_brownien(t,n)` permettant d'obtenir une réalisation de $X_n(t)$; dans la suite on appelera cette fonction m fois avec une valeur de n assez grande[18] il faut donc l'écrire sans utiliser de boucle.
2. Ecrire un script scilab mettant en place cette simulation du pont Brownien : effectuer m simulations de $X_n(t)$ (avec n grand) et illustrer graphiquement la convergence en loi (en dessinant la fonction de répartition empirique et en lui juxtaposant la fonction de répartition exacte de $Y(t)$ puis effectuer le test de Kolmogorov-Smirnov. On pourra placer la fonction précédente en début du script pour travailler avec un seul fichier.

[18] pour essayer d'approcher $Y(t)$!

Remarque : il n'est pas facile de régler de bonnes valeurs pour m et n : lorsque m croit il y a un moment où le test échoue car il détecte en fait que n n'est pas assez grand (puisque la loi normale est obtenue à la limite lorsque $n \to +\infty$), il faut alors réaugmenter n... Avec $t = 0.3$, vous pouvez par exemple tester avec $n = 1000$ et $m = 200, 500, 1000$, et le test fonctionne bien (il y a très peu de rejet) mais avec $m = 10000$ le test est presque toujours négatif. Si on augmente n (avec par exemple $n = 10000$ mais les calculs commencent à être un peu long sur un PC performant de 2003) alors le test redevient « normalement » positif.

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(k) = quelque chose
   y(k) = 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 `x` 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
endfunction
```

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;
```

Remarque : ce problème est résolu dans la version de développement (cvs) de Scilab.

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` (ou `exec`) ! Une petite astuce : votre instruction `getf` (ou `exec`) 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 Nombres Complexes et nombres réels

Tout est fait dans Scilab pour traiter de la même manière les réels et les complexes ! Ceci est assez pratique mais peut conduire à certaines surprises, par exemple lorsque vous évaluez une fonction réelle hors de son domaine de définition (par exemple $\sqrt{x}$ et $\log(x)$ pour $x < 0$, $\text{acos}(x)$ et $\text{asin}(x)$ pour $x \notin [-1, 1]$, $\text{acosh}(x)$ pour $x < 1$) car Scilab renvoie alors l'évaluation de l'extension complexe de la fonction. Pour savoir si l'on a affaire a une variable réelle ou complexe, on peut utiliser la fonction `isreal` :

```
-->x = 1
 x  =
    1.
```

```
-->isreal(x)
 ans  =
  T

-->c = 1 + %i
 c  =
    1. + i

-->isreal(c)
 ans  =
  F

-->c = 1 + 0*%i
 c  =
    1.

-->isreal(c)
 ans  =
  F
```

6.8 Primitives et fonctions Scilab

Dans ce document j'ai utilisé plus ou moins indifféremment les termes *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. Depuis la version 2.7, les primitives sont aussi des sortes de variables Scilab (de type fptr). Néanmoins il subsiste quelques problèmes (actuellement résolus dans la version de développement).

Voici un exemple de problème que l'on peut rencontrer : la fonction suivante permet d'approcher une intégrale par la méthode de *Monte Carlo* :

```
function [im,sm]=MonteCarlo(a,b,f,m)
   //            /b
   //  approx de |  f(x) dx par la methode de Monte Carlo
   //            /a
   //  on renvoie aussi l'ecart type empirique
   xm = grand(m,1,"unf",a,b)
   ym = f(xm)
   im = (b-a)*mean(ym)
   sm = (b-a)*st_deviation(ym)
endfunction
```

Pour l'argument f, une fonction Scilab est attendue mais ce code devrait aussi fonctionner avec les fonctions mathématiques qui sont des primitives Scilab[1] puisqu'elles sont maintenant considérées comme des variables. Néanmoins un test avec une primitive, par exemple `exp` échoue :

```
-->[I,sigma]=MonteCarlo(0,1,exp,10000)  // bug !
```

Cependant si vous chargez la fonction `MonteCarlo` avec `getf` muni de l'option de non compilation :

```
-->getf("MonteCarlo.sci","n")
```

ce bug (corrigé dans le cvs actuel) est alors contourné.

[1] par exemple `sin`, `exp` et beaucoup d'autres sont des primitives.

6.9 Évaluation d'expressions booléennes

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

a ou b
a 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 »).

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. Un script possible (à expérimenter avec différentes valeurs de n) :
```
n = 2000;
x = rand(1,n);
xbar = cumsum(x)./(1:n);
plot(1:n, xbar, "n","xbar", ...
    "illustration de la lgn : xbar(n) -> 0.5 qd n -> + oo")
```

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
endfunction
```

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
endfunction
```

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
endfunction
```

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
endfunction
```

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
endfunction

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
endfunction
```

6. La rencontre : soit T_A et T_B les temps d'arrivée de Mr A et Melle B. Ce sont deux variables aléatoires indépendantes de loi $U([17,18])$ et la rencontre à lieu si $[T_A, T_A+1/6] \cup [T_B, T_B+1/12] \neq \emptyset$. En fait une expérience de la rencontre correspond à la réalisation de la variable aléatoire vectorielle $R = (T_A, T_B)$ qui, avec les hypothèses, suit une loi uniforme sur le carré $[17,18] \times [17,18]$. La probabilité de la rencontre se calcule donc en faisant le rapport entre la surface de la zone (correspondant à une rencontre effective) définie par :

$$\begin{cases} T_A \leq T_B + 1/12 \\ T_B \leq T_A + 1/6 \\ T_A, T_B \in [17,18] \end{cases}$$

et la surface du carré (1), ce qui permet d'obtenir $p = 67/288$. Pour calculer cette probabilité, on peut bien sûr faire comme si la rencontre devait avoir lieu entre minuit et une heure :-) ! Par simulation, on effectue m expériences et la probabilité empirique est le nombre de cas où la rencontre a lieu divisé par m. Voici une solution :

```
function [p] = rdv(m)
   tA = rand(m,1)
   tB = rand(m,1)
   rencontre = tA+1/6 > tB  &  tB+1/12 > tA;
   p = sum(bool2s(rencontre))/m
endfunction
```

on peut aussi utiliser la fonction `grand` décrite dans le chapitre sur les applications :

```
function [p] = rdv(m)
   tA = grand(m,1,"unf",17,18)
   tB = grand(m,1,"unf",17,18)
   rencontre = tA+1/6 > tB  &  tB+1/12 > tA;
   p = sum(bool2s(rencontre))/m
endfunction
```

Annexe C

Correction des exercices du chapitre 4

Dé truqué ou non ?

Si on note J la variable aléatoire correspondant au résultat de l'expérience alors J suit la loi géométrique $G(p)$ avec $p = 1/6$ si le dé est non truqué. Avec les données de l'expérience, on agglomère dans une même classe tous les résultats correspondants à un nombre de lancés supérieur strictement à 10. Ce qui nous donne les probabilités (notant $q = 1 - p = 5/6$) :

$$P(J = 1) = p,\ P(J = 2) = qp,\ P(J = 3) = q^2p,\ ....,\ P(J = 10) = q^9p,\ P(J > 10) = q^{10}$$

D'autre part le tableau donne directement le nombre d'occurence, d'où le script :

```
occ= [36 ; 25 ; 26 ; 27 ; 12 ; 12 ; 8 ; 7 ; 8 ; 9 ; 30];
p = 1/6; q = 5/6;
pr = [p*q.^(0:9) , q^10]';
y = sum( (occ - m*pr).^2 ./ (m*pr) );
y_seuil = cdfchi("X", 10, 0.95, 0.05);
mprintf("\n\r Test sur le de :")
mprintf("\n\r y = %g, et y_seuil (95 \%) = %g", y, y_seuil)
mprintf("\n\r 200*min(pr) = %g", 200*min(pr))
```

On obtient $y = 7.73915$ alors que $y_{seuil} = 18.307$, ce dé semble donc correct. Néanmoins on a $200 \times \min(pr) \simeq 6.5$ ce qui est tout juste au dessus de la condition d'applicabilité du test (on pourrait donc demander quelques expériences supplémentaires).

Urne de Polya

La fonction scilab

```
function [XN, VN] = Urne_de_Polya_parallele(N,m)
   VN = ones(m,1) ; V_plus_R = 2 ; XN = 0.5*ones(m,1)
   for i=1:N
      u = grand(m,1,"de"f)     // 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 dans lesquelles
                               // on a tire une boule 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
endfunction
```

Le script Scilab

```
// polya simulation :
N = 10;
m = 5000;
[XN, VN] = Urne_de_Polya_parallele(N,m);
```

```
// 1/ calcul de l'espérance et de l'intervalle de securite
EN = mean(XN);            // esperance empirique
sigma = st_deviation(XN); // ecart type empirique
delta =  2*sigma/sqrt(m); // increment pour l'intervalle empirique (2 pour 1.9599..)
mprintf("\n\r  E exact = 0.5");
mprintf("\n\r  E estime = %g",EN);
mprintf("\n\r  Intervalle (empirique) de confiance a 95\% : [%g,%g]",EN-delta,EN+delta);

// 2/ test chi2
alpha = 0.05;
p = 1/(N+1);  //  proba theorique pour chaque resultat (loi uniforme)
occ = zeros(N+1,1);
for i=1:N+1
  occ(i) = sum(bool2s(XN == i/(N+2)));
end
if sum(occ) ~= m then, error(" Probleme..."), end  // petite verification
Y = sum( (occ - m*p).^2 / (m*p) );
Y_seuil = cdfchi("X",N,1-alpha,alpha);
mprintf("\n\r Test du chi 2 : ")
mprintf("\n\r -------------- ")
mprintf("\n\r valeur obtenue par le test : %g", Y);
mprintf("\n\r valeur seuil a ne pas depasser : %g", Y_seuil);
if (Y > Y_seuil) then
  mprintf("\n\r Conclusion provisoire : Hypothese rejetee !")
else
  mprintf("\n\r Conclusion provisoire : Hypothese non rejetee !")
end

// 3/ illustrations graphiques
function [d] = densite_chi2(X,N)
   d = X.^(N/2 - 1).*exp(-X/2)/(2^(N/2)*gamma(N/2))
endfunction
frequences = occ/m;
ymax = max([frequences ; p]); rect = [0 0 1 ymax*1.05];
xbasc(); xset("font",2,2)
subplot(2,1,1)
   plot2d3((1:N+1)'/(N+2), frequences, style=2 , ...
   frameflag=1, rect=rect, nax=[0 N+2 0 1])
   plot2d((1:N+1)'/(N+2),p*ones(N+1,1), style=-2, strf="000")
   xtitle("Frequences empiriques (traits verticaux) et probabilites exactes (croix)")
subplot(2,1,2)
   // on trace la densite chi2 a N ddl
   X = linspace(0,1.1*max([Y_seuil Y]),50)';
   D = densite_chi2(X,N);
   plot2d(X,D, style=1, leg="densite chi2", axesflag=2)
   // un trait vertical pour Y
   plot2d3(Y, densite_chi2(Y,N), style=2, strf="000")
   xstring(Y,-0.01, "Y")
   // un trait vertical pour Yseuil
   plot2d3(Y_seuil, densite_chi2(Y_seuil,N), style=5, strf="000")
   xstring(Y_seuil,-0.01, "Yseuil")
   xtitle("Positionnement de Y  par rapport a la valeur Yseuil")
```

Voici un résultat obtenu avec $N = 10$ et $m = 5000$ (cf figure (C.1)) :

```
 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 !
```

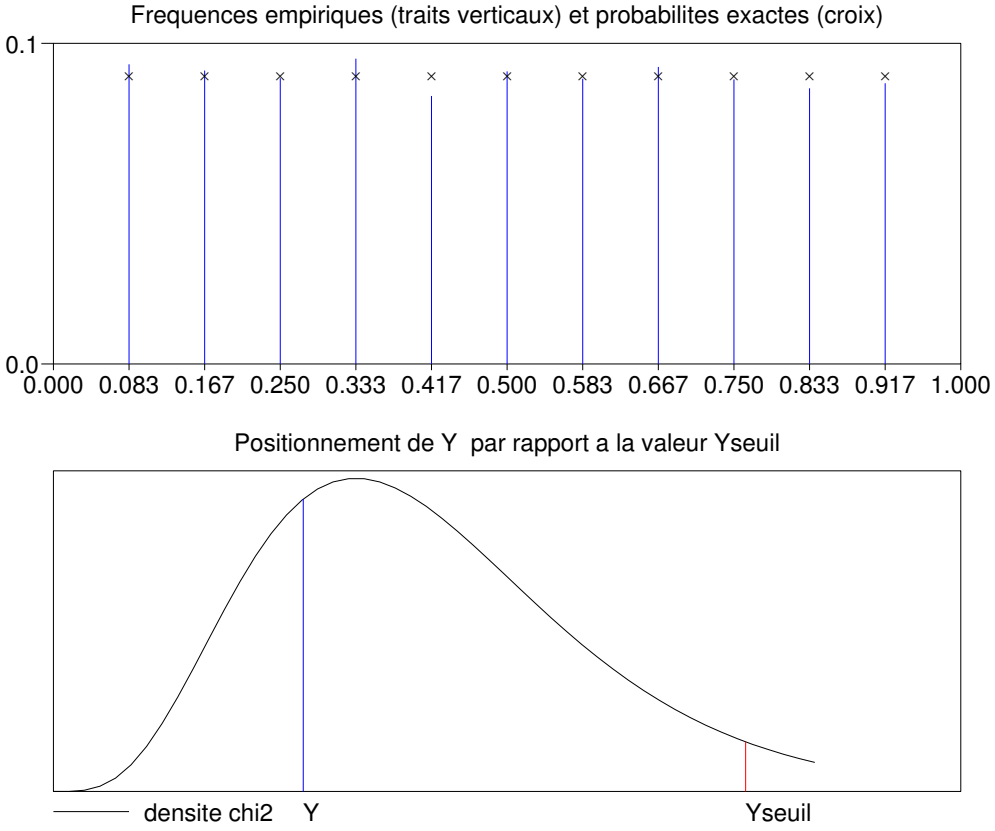

FIG. C.1 – Illustrations pour le test du χ^2 de l'urne de Polya...

Le pont brownien

La fonction

```
function [X] = pont_brownien(t,n)
   X = sum(bool2s(grand(n,1,"def") <= t) - t)/sqrt(n)
endfunction
```

Le script

Ce script effectue m simulations de la variable aléatoire $X_n(t)$, affiche le graphe de la fonction de répartition empirique, lui superpose celui de la fonction de répartition attendue (pour $n = +\infty$) et finalement effectue le test KS :

```
t = 0.3;
sigma = sqrt(t*(1-t));  // l'écart type attendu
n = 4000;               // n "grand"
m = 2000;               // 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

// le dessin de la fonction de repartition empirique
X = - sort(-X); // tri
prcum = (1:m)'/m;
```

```
xbasc()
plot2d2(X, prcum)
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,style=2, strf="000")                 // on l'ajoute sur le dessin initial

// mise en place du test KS
alpha = 0.05;
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(0.5*log(2/alpha));

// affichage des resultats
//
mprintf("\n\r Test KS : ")
mprintf("\n\r -------- ")
mprintf("\n\r  valeur obtenue par le test : %g", Km);
mprintf("\n\r  valeur seuil a ne pas depasser : %g",K_seuil);
if (Km > K_seuil) then
   mprintf("\n\r Conclusion provisoire : Hypothese rejetee !")
else
   mprintf("\n\r Conclusion provisoire : Hypothese non rejetee !")
end
```

Voici des résultats obtenus avec $n = 1000$ et $m = 4000$:

```
Test KS :
--------
valeur obtenue par le test : 1.1204036
valeur seuil a ne pas depasser : 1.2212382
Conclusion provisoire : Hypothese non rejetee !
```

Index