Scilab: une introduction
Version 1.0

J.Ph. Chancelier[1]

31 août 2004

[1]Cermics, École Nationale des Ponts et Chaussées, 6 et 8 avenue Blaise Pascal, 77455, Marne la Vallée, Cedex 2

Table des matières

Première partie

Une introduction à Scilab

Chapitre 1

Une introduction à Scilab

1.1 Présentation du langage Scilab

Scilab est un langage interprété, à typage dynamique, extensible et gratuit. Il a d'abord été développé sous le nom de Basile (Inria projet Meta2) puis sous le nom de Scilab par le Scilab Group (Chercheurs du projet Inria Metalau et de l'Enpc Cermics). Son développement actuel est coordonné par un Consortium et un nombre non négligeable de contributions proviennent de contributeurs extérieurs. Scilab est un langage portable. Il est porté sur les différentes variantes d'UNiX, mais aussi sur Windows et MacOS X.

On peut considérer que Scilab est un langage de script puisque il peut servir à une illustration algorithmique en quelques lignes. Mais c'est aussi un langage de programmation et la librairie Scilab contient presque 100000 lignes de code de fonctions écrites en Scilab. La syntaxe de Scilab est simple, les données propres au calculs scientifique (Matrices) y sont d'un maniement aisé. Cela conduit a des programmes compacts et lisibles souvent beaucoup courts que des programmes C, C++ ou Java équivalents.

Scilab est avant tout dédié au calcul scientifique. Il permet d'accéder de façon aisée à de nombreuses librairies numériques : calcul matriciel, intégration numérique, optimisation, équations différentielles, ... Scilab est extensible. On peut facilement rajouter de nouvelles fonctionnalités a Scilab avec des librairies extérieures existantes. De nombreuses contributions qui utilisent cette possibilité sont disponibles sur le site officiel de Scilab :

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

Un site Scilab à l'Enpc contient de nombreux travaux pratiques illustrant les cours de mathématiques proposée a l'Enpc :

`http://cermics.enpc.fr/scilab/`

La fenêtre principale de Scilab (dans la version Unix Gtk) se présente comme sur la figure 1.3. La visualisation des résultats numériques peut se faire grâce à une bibliothèque graphique utilisable interactivement (Voir Figure 1.1). Enfin, les aides de toutes les fonctions sont accessibles en ligne et elles comportent généralement un exemple de code Scilab illustrant leur utilisation. Les deux commandes les plus importantes de Scilab sont sans doute `help` et `apropos`, elles permettent de naviguer dans l'aide. De nombreuses demonstrations sont aussi disponibles en lignes (Menu `demos`).

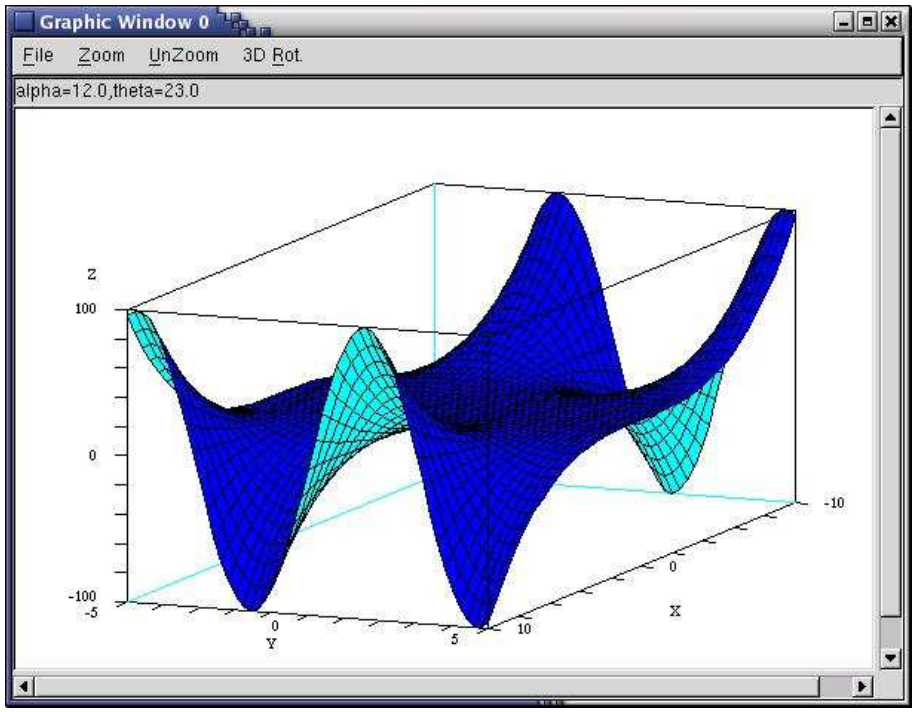

FIG. 1.1 – Une fenêtre graphique

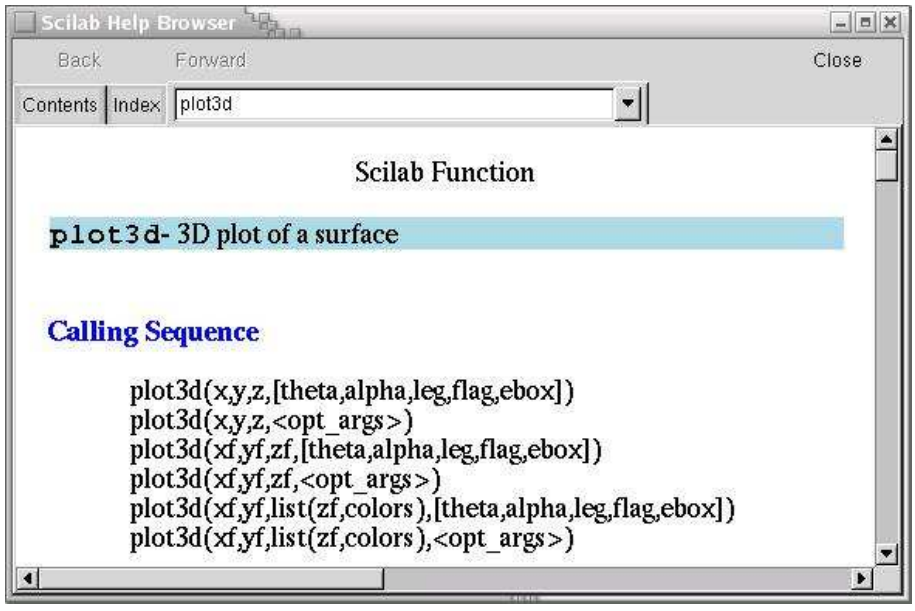

FIG. 1.2 – Une fenêtre de l'aide en ligne

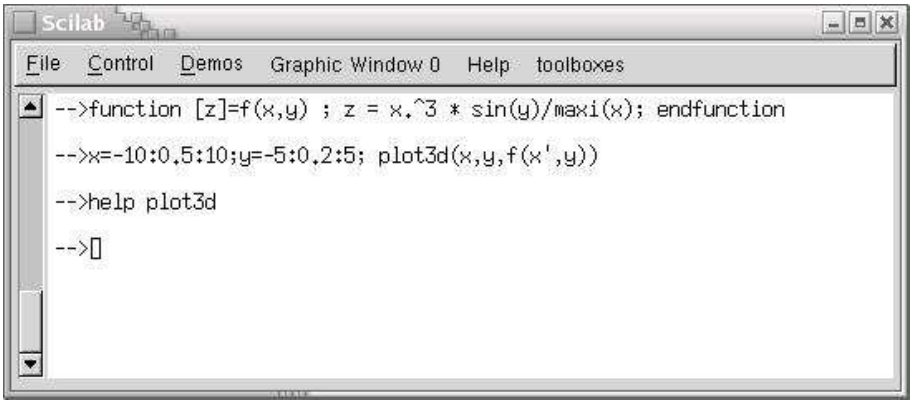

FIG. 1.3 – La fenêtre principale

1.2 Quelques idées générales

Le langage Scilab est un langage qui a été initialement conçu pour privilégier les opérations matricielles en ciblant des applications scientifiques. Les objets ne sont jamais déclarés ou alloués explicitement, ils ont un type dynamique (renvoyé par la fonction `typeof`) et leur taille peux changer dynamiquement au gré des opérateurs qui leurs sont appliqués :

```
-->a= rand(2,3);                          ← a est (après exécution de =) une matrice 2x3 scalaire
-->typeof(a)                              ← type de a ? constant désigne les matrices de réels ou de com-
plexes (en double précision)
 ans  =

 constant

-->a= [a, zeros(2,1)]                                          ← la taille de a augmente
 a  =

!   0.2113249    0.0002211    0.6653811    0. !
!   0.7560439    0.3303271    0.6283918    0. !

-->a= 'scilab'; typeof(a)                          ← nouveau type de a par réaffectation.
 ans  =

 string

-->exists('a')                                                 ← a est-elle définie ?
 ans  =

    1.

-->clear('a'); exists('a')                                ← destruction explicite de a
 ans  =

    0.
```

La mémoire disponible à un moment donné par l'utilisateur peut-être contrôlée (c'est-à-dire demandée ou changée) au moyen de la commande `stacksize`. Elle est bien sûr limitée par la mémoire dynamique que l'on peut demander (mémoire allouée par `malloc` sur le "tas"). Les commandes `who` et `whos` permettent d'obtenir les noms et les tailles des variables présentes dans l'environnement d'exécution.

```
-->stacksize()          ← mémoire disponible en nombre de doubles (8 octets) et nombre maximum de variables
 ans  =

!   1000000.    5336. !

-->A=rand(1000,1000);                                   ← noter qu'une expression terminée par ; est évaluée mais le résultat n'est pas affiché
                  !--error    17
rand: stack size exceeded (Use stacksize function to increase it)


-->stacksize(6000000);                                  ← modification de la taille mémoire disponible
-->A=rand(1000,1000);                                   ← On peut maintenant générer A
```

L'évaluation des expressions Scilab se fait au moyen d'un interprète appelé `scilab`. C'est l'interprète qui gère le contexte d'exécution des expressions Scilab.

Dans le contexte d'exécution de Scilab, un utilisateur dispose par défaut d'un grand nombre de fonctions. Ces fonctions peuvent être des fonctions écrites en Scilab ; nous verrons en effet que Scilab est un langage de programmation. Mais elle peuvent aussi avoir été écrites en `C`, `Fortran`, ... et rendues accessibles au niveau de l'interprète Scilab.

Ce sont ces nombreuses fonctions facilement utilisables qui font la richesse et la puissance du langage. Notons aussi qu'un utilisateur peut enrichir l'environnement de base et cela de façon dynamique avec de nouvelles fonctions Scilab qui peuvent utiliser du code de librairies externes `C`, `Fortran`, ... On peut donc développer des "boîtes à outils" pour enrichir l'environnement d'exécution de Scilab et l'adapter à des besoins spécifiques.

1.3 Les objets

Comme il a déjà été dit, Scilab a été conçu au départ pour le calcul scientifique et l'objet de base est la matrice de nombres flottants en double précision, un scalaire étant un cas particulier de matrice de taille `1x1`. Par défaut les nombres utilisés par Scilab sont donc des matrices de `double`.

```
-->a=1:0.6:3                            ← a est maintenant une matrice scalaire (de double)
 a  =

!   1.    1.6    2.2    2.8 !

-->b=[%e,%pi]                   ← b est le vecteur 1x2 contenant deux valeurs prédéfinies (e, π)
```

```
 b  =

!   2.7182818    3.1415927 !

-->b(a)                                   ← a peut en fait être utilisé comme indice entier
 ans  =

!   2.7182818    2.7182818    3.1415927    3.1415927 !
```

Au cours du temps Scilab s'est enrichi de nouveaux types en privilégiant l'aspect matriciel : matrices de chaînes de caractères, matrices de booléens, matrices creuses (de scalaires ou de booléens), matrices d'entiers (`int8`, `int16`, `int32`), matrice de fractions rationnelles et du type `list` (plus quelques variantes) permettant de construire en fait des structures :

```
-->a=["A","B";"C","D"]                    ← Une matrice de chaînes de caractères
 a  =

!A  B  !
!      !
!C  D  !

-->b=rand(2,2)                            ← Une matrice de scalaires
 b  =

!   0.2113249    0.0002211 !
!   0.7560439    0.3303271 !

-->b= b>= 0.5                             ← Une matrice de booléens
 b  =

! F F !
! T F !

-->l=list(a,b)                            ← Une liste d'objets
 l  =

       l(1)

!A  B  !
!      !
!C  D  !

       l(2)

! F F !
! T F !

-->a= spec(rand(3,3))                     ← Un vecteur de nombres complexes
 a  =

!   1.8925237              !
```

```
!   0.1887123 + 0.0535764i !
!   0.1887123 - 0.0535764i !
```

Il est possible de définir de nouveaux “types” d'objets en Scilab, au sens où il est possible de définir des objets dont le type dynamique (valeur retournée par `typeof`) est défini par l'utilisateur. Les opérateurs de Scilab (par exemple l'opérateur `+`) peuvent être surchargés pour tenir compte de ces nouveaux type dynamiques.

```
-->function [y]=eleve(nom,promo)                                    ← Une fonction Scilab
-->  [y]=tlist(['Eleve','nom','promo'],nom,promo)                   ← tlist est uti-
lisé pour construire une structure en spécifiant son type dynamique et les noms de ses champs.
-->endfunction
-->el=eleve('Lapeyre',64);                                          ← Un objet de  type 'Eleve'
-->typeof(el)
 ans  =

 Eleve

-->function %Eleve_p(el)   ← Surcharge de la fonction d'impression (indiqué par le suffixe ''_p'')
-->  mprintf('Nom: %s, promo: %d\n',el.nom,el.promo);
-->endfunction
-->el                                                               ← test de la fonction d'impression
 el  =

 Nom: Lapeyre, promo: 64

-->el.nom                                                           ← accès à un champ
 ans  =

 Lapeyre
```

1.4 Objets vectoriels et matriciels

Comme nous l'avons déjà souligné, une des caractéristique de Scilab est de privilégier les opérations matricielles quel que soit le type des éléments des matrices. Nous passons en revue dans cette section les opérateurs généraux de construction d'objets matriciels communs à tous les types de matrices.

Le type matrice dans Scilab désigne en fait des tableaux à au plus deux indices qui sont de façons interne codés comme des tableaux unidimensionnels. On peut donc toujours accéder aux éléments d'une matrice avec un ou deux indices, le stockage des éléments étant fait “colonne par colonne”. Les vecteurs et scalaires sont en Scilab traités comme des cas particuliers de matrices.

Pour construire des tableaux multidimensionnels en Scilab on dispose du type `hypermat`. Ce type est implémenté en Scilab (du moins jusque dans la version 2.7) et les opérateurs associés sont en fait beaucoup moins performants que les opérateurs sur les matrices “usuelles”.

Les opérateurs élémentaires de construction qui sont surchargés pour tous les types de matrices sont l'opérateur de concaténation en ligne “;” et l'opérateur de concaténation en

colonne ",". Ces deux opérateurs ne sont considérés comme des opérateurs de concaténation que, dans un contexte matriciel, (i.e entre les symboles "[" et "]"). Tout leurs arguments doivent avoir le même type. On notera que dans un contexte matriciel le symbole blanc est synonyme de "," et le retour à la ligne synonyme de ";" et que cela peut être source de confusion :

```
-->A=[1,2,3 +5]                          ← Attention ici A=[1,2,3,+5] avec un + unaire
 A  =

!   1.    2.    3.    5. !

-->A=[1,2,3 *5]                          ← Attention ici A=[1,2,3*5] avec un * binaire
 A  =

!   1.    2.    15. !

-->A=[A,0; 1,2,3,4]
 A  =

!   1.    2.    15.    0. !
!   1.    2.    3.     4. !
```

Le tableau 1.1 décrit des fonctions de construction de matrices utilisées fréquemment.

`diag`	matrice `(m,n)` à diagonale fixée (ou extraction de diagonales)
`eye`	matrice `(m,n)` avec des uns sur la diagonale principale
`grand`	matrice `(m,n)` scalaire aléatoire
`linspace` ou " : "	vecteur à valeurs régulièrement espacées
`logspace`	vecteur à valeurs logarithmiquement espacées
`matrix`	changement de dimensions `(m,n)` à produit `m*n` fixé
`ones`	matrice `(m,n)` de uns
`rand`	matrice `(m,n)` scalaires aléatoires (lois uniforme ou Gaussienne)
`zeros`	matrice `(m,n)` de zéros
`.*.`	opérateur de Kronecker

TAB. 1.1 – Quelques fonctions de construction de matrices

Et nous en donnons une illustration rapide :

```
-->A= [eye(2,2), 3*ones(2,3); linspace(3,9,5); zeros(1,5)]
 A  =

!   1.    0.     3.    3.     3. !
!   0.    1.     3.    3.     3. !
!   3.    4.5    6.    7.5    9. !
!   0.    0.     0.    0.     0. !
```

```
-->d=diag(A)                                    ← extraction de la diagonale principale
 d  =

!   1. !
!   1. !
!   6. !
!   0. !

-->B=diag(d)                                    ← construction d'une matrice diagonale
 B  =

!   1.    0.    0.    0. !
!   0.    1.    0.    0. !
!   0.    0.    6.    0. !
!   0.    0.    0.    0. !

-->C=matrix(d,2,2)                              ← changement des dimensions
 C  =

!   1.    6. !
!   1.    0. !

-->D=[1,2,3].*.ones(2,2)                        ← produit de Kronecker
 D  =

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

La plupart des fonctions Scilab acceptent des arguments de type matriciel et il faut se reporter à la documentation de chaque opérateur pour en comprendre la sémantique :

```
-->A=rand(2,2)                                  ← matrice aléatoire avec loi uniforme par défaut
 A  =

!   0.2113249    0.0002211 !
!   0.7560439    0.3303271 !

-->B=exp(A)                                     ← exponentiation termes à termes B(i,j)=exp(A(i,j))
 B  =

!   1.2353136    1.0002212 !
!   2.1298336    1.3914232 !

-->B=expm(A)                                    ← exponentiation matricielle
 B  =

!   1.2354211    0.0002901 !
!   0.9918216    1.391535  !
```

1.4.1 Extraction, insertion et délétion

Pour désigner des éléments d'une matrice, on utilise la notation `A(B)` ou `A(B,C)` ; où `B` et `C` sont des matrices d'indices ou des matrices de booléens.

Quand une expression `A(B)` ou `A(B,C)` est un membre gauche d'une affectation, alors il s'agit d'une opération d'affectation si le membre de droite de l'affectation s'évalue à une matrice non vide. On donne alors une valeur à la sous matrice désignée par le membre gauche en utilisant la valeur du membre droit. Il faut bien sûr que les deux sous matrices aient des tailles compatibles (i.e il faut qu'elles aient la même taille ou bien que le membre droit soit scalaire).

Il peut aussi s'agir d'une opération de délétion si le membre droit s'évalue à une matrice vide. On élimine alors les éléments désignés par le membre gauche. Les opérations d'affectation et de délétion changent bien sûr dynamiquement les dimensions de la matrice concernée.

```
-->clear A;
-->A(2,4) = 1
 A  =

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

-->A([1,2],[1,2])=int(5*rand(2,2))          ← affectation qui change une sous matrice de A
 A  =

!   1.    0.    0.    0. !
!   3.    1.    0.    1. !

-->A([1,2],[1,3])=[]                                ← délétion d'une sous matrice
 A  =

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

-->A(:,1)= 8   ← affectation ":" désigne l'ensemble des indices possibles, ici toutes les lignes de A
 A  =

!   8.    0. !
!   8.    1. !

-->A(:,$)=[]                                ← délétion "$" désigne le dernier indice
 A  =

!   8. !
!   8. !

-->A(:,$+1)=[4;5]                                ← rajout d'une colonne par affectation
 A  =

!   8.    4. !
!   8.    5. !
```

Quand une expression `A(B)` ou `A(B,C)` apparaît dans une expression Scilab (hors membre gauche d'une affectation), elle désigne alors une sous matrice de la matrice `A` et son évaluation conduit à la création d'une nouvelle matrice.

```
-->A = int(10*rand(3,7))                                    ← int calcule la partie entière
 A  =

!   2.    3.    8.    0.    7.    2.    8. !
!   7.    6.    6.    5.    1.    2.    6. !
!   0.    6.    8.    6.    5.    2.    3. !

-->B=A([1,3],$-1:$)                                         ← extraction d'une sous ma-
trice (lignes un et trois; avant dernière et dernière colonnes)
 B  =

!   2.    8. !
!   2.    3. !
```

Une expression `A(B)` ou `A(B,C)` désigne une sous matrice de la matrice `A`. Quand `B` et `C` sont des matrices scalaires elles désignent les indices à utiliser pour désigner la sous matrice. Quand `B` et `C` sont des matrices de booléens, alors les indices à considérer sont les indices pour lesquels `B` et `C` prennent la valeur `T`. On peut utiliser la fonction `find` pour obtenir explicitement les indices correspondants.

```
-->A = int(10*rand(2,4))                                    ← int calcule la partie entière
 A  =

!   2.    0.    6.    8. !
!   7.    3.    6.    6. !

-->A( A>= 5) = []      ← délétion : les indices sont donnés par une matrice de booléens A>=5. No-
ter que A devient une matrice colonne.
 A  =

!   2. !
!   0. !
!   3. !

-->I=find(A>=2)                                             ← indices des éléments de A qui sont >=2
 I  =

!   1.    3. !
```

1.4.2 Scalaires et matrices de scalaires

Les matrices de scalaires désignent dans Scilab les matrices dont les éléments sont des nombres flottants en double précision (`double`) ou bien des nombres complexes en double

précision (deux `double`). Leur type est désigné par `constant` (valeur retournée par `typeof`). La norme adoptée sur les calculs en double précision peut être contrôlée par la fonction `ieee` :

```
-->ieee()                                          ← le mode de calcul par défaut
 ans  =

    0.

-->1/0                              ← Dans le mode par défaut 1/0 provoque une erreur Scilab
    !--error    27
division by zero...

-->ieee(2)                                ← le mode par défaut est maintenant le mode ieee
-->1/0                                          ← 1/0 est maintenant évalué a  + ∞
 ans  =

   Inf
```

Les calculs se font avec une certaine précision qui dépend de la machine sur laquelle Scilab est utilisé. La variable Scilab prédéfinie `%eps`, donne cette précision machine. C'est le plus grand nombre en double précision tel que `1+(%eps)/2 ==1`.

```
-->x = [1.32,2,2.34e-3,%inf,%pi,%s,%nan,1+%eps]          ← quelques scalaires Scilab
 x  =

!   1.32     2     0.00234    Inf     3.1415927     s    Nan    1  !
```

C'est bien sûr pour le type `constant` que l'on trouve un très grand nombre d'opérateurs et de fonctions matricielles prédéfinis.

Nous donnons ici les principaux opérateurs Scilab et les règles de précédence qui les régissent. Il sont tous définis par défaut pour les matrices de scalaires (même les opérateurs logiques) et sont surchargés ou surchargeables pour les autres types de données. Les opérateurs sont classés en ligne par ordre de précédence croissante. Au sein d'un même ligne, les opérateurs ont même précédence avec associativité à gauche sauf pour les opérateurs puissance ou l'associativité est a droite.

Les opérateurs commençant par le symbole . désignent en général des opérateurs agissant termes à termes. Par exemple `C=A.*B` calcule la multiplication termes à termes des deux matrices `A` et `B` à savoir `C(i,j)=A(i,j)*B(i,j)`.

```
-->A=(1:9)' * ones(1,9)                  ← transposition ' et produit matriciel usuel *
 A  =

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

`|`	ou logique
`&`	et logique
`~`	non logique
`==, >=, <=, >, < ,<>, ~=`	opérateurs de comparaison
`+,-`	addition et soustraction binaire
`+,-`	addition et soustraction unaires
`.*, ./, .\, .*., ./., .\., *, /, /., \.`	"multiplications" et "divisions"
`^,** , .^ , .**`	puissances
`', .'`	transpositions

TAB. 1.2 – Précédences des opérateurs

```
!   5.    5.    5.    5.    5.    5.    5.    5.    5. !
!   6.    6.    6.    6.    6.    6.    6.    6.    6. !
!   7.    7.    7.    7.    7.    7.    7.    7.    7. !
!   8.    8.    8.    8.    8.    8.    8.    8.    8. !
!   9.    9.    9.    9.    9.    9.    9.    9.    9. !

-->A.* A'                          ← révisons nos tables de multiplications : produit termes à termes
 ans  =

!   1.    2.     3.     4.     5.     6.     7.     8.     9.  !
!   2.    4.     6.     8.     10.    12.    14.    16.    18. !
!   3.    6.     9.     12.    15.    18.    21.    24.    27. !
!   4.    8.     12.    16.    20.    24.    28.    32.    36. !
!   5.    10.    15.    20.    25.    30.    35.    40.    45. !
!   6.    12.    18.    24.    30.    36.    42.    48.    54. !
!   7.    14.    21.    28.    35.    42.    49.    56.    63. !
!   8.    16.    24.    32.    40.    48.    56.    64.    72. !
!   9.    18.    27.    36.    45.    54.    63.    72.    81. !

-->t=(1:4)';m=size(t,'r');n=4;
-->A=(t*ones(1,n+1)).^( ones(m,1)*[0:n])                                    ← puis-
sances termes à termes pour construire une matrice de Vandermonde  A(i,j) = t_i^{j-1}
 A  =

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

-->A=eye(2,2).*.[1,2;3,4]                                  ← produit de Kronecker
 A  =

!   1.    2.    0.    0. !
!   3.    4.    0.    0. !
!   0.    0.    1.    2. !
!   0.    0.    3.    4. !

-->A=[1,2;3,4];b=[5;6]; x = A \ b ; norm(A*x -b)  ← \ résolution du système linéaires Ax=b
 ans  =
```

```
    0.

-->A1=[A,zeros(A)]; x = A1 \ b                                    ← Système sous-
déterminé : on obtient une solution particulière
 x  =

! - 4.  !
!   4.5 !
!   0.  !
!   0.  !

-->A1=[A;A]; x = A1\ [b;7;8]      ← Système sur-déterminé : solution au sens des moindres carrés
 x  =

! - 5.  !
!   5.5 !
```

1.4.3 Chaînes de caractères

Les chaînes de caractères Scilab sont délimitées par les caractères apostrophe “ ’ ” ou guillemets anglo-saxons “ " ” (qui sont équivalents). Pour insérer une apostrophe ou des guillemets, il faut les faire précéder d’un délimiteur (à nouveau “ ’ ” ou “ " ”). Les opérations de base sur les chaînes de caractères sont la concaténation, notée par l’opérateur “ + ” et la fonction `length` qui renvoie le nombre de caractères contenus dans une chaîne. On peut bien sur créer des matrices de chaînes de caractères et les deux opérateurs précédemment décrits deviennent alors vectoriels et les opérateurs usuels de construction de matrices par concaténation de colonnes ou de lignes sont utilisables :

```
-->S="une chaîne avec <<"’>> "
 S  =

 une chaîne avec <<’>>

-->S=’une chaîne beaucoup plus longue...          ← ... pour continuer sur la ligne suivante
--> avec un mot en ""guillemets"" "
 S  =

 une chaîne beaucoup plus longue avec un mot en "guillemets"

-->S=[’Une’,’de’;’matrice’,’chaînes’]                          ← une matrice de chaînes
 S  =

!Une      de       !
!                  !
!matrice  chaînes  !

-->length(S)                                  ← longueur de chaque chaîne de la matrice S
 ans  =
```

```
!   3.    2. !
!   7.    7. !
```

`ascii`	conversion entre chaîne et codes Ascii
`execstr`	fait exécuter par l'interprète une chaîne de caractère
`grep`	recherche de sous-chaînes dans une matrice de chaînes
`part`	extraction de sous-chaînes
`msscanf`	lecture formatée (C) dans un chaîne
`msprintf`	construction d'un vecteur ou d'une chaîne par écriture formatée
`strindex`	position d'occurrences de sous chaînes dans une chaîne
`string`	conversion vers une chaîne de caractère
`stripblanks`	enlève les caractères blancs initiaux et finaux
`strsubst`	substitution de sous-chaînes
`tokens`	découpe une chaîne
`strcat`	concatène les éléments d'une matrice
`length`	longueur des chaînes d'une matrice de chaîne

TAB. 1.3 – Quelques fonctions sur les chaînes de caractères

Dans le tableau 1.3, sont répertories plusieurs fonctions utiles liées à l'utilisation des chaînes de caractères. Nous les illustrons maintenant au moyen d'un exemple. On cherche ici à transformer une matrice scalaire en une chaîne de caractère permettant d'obtenir une représentation en LaTeX de cette matrice. Connaître le langage LaTeX n'a ici aucune importance, il s'agit, partant de la matrice `A=testmatrix('magi',3)`, de construire la chaîne de caractère :

```
\[ A= \begin{array}{ccc} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \\ \end{array} \]
```

```
-->A=testmatrix('magi',3)                                  ← un carré magique 3x3
 A  =

!   8.    1.    6. !
!   3.    5.    7. !
!   4.    9.    2. !

-->[ma,na]=size(A);                                        ← taille de la matrice de chaînes
-->B=string(A)                  ← transformation d'une matrice scalaire en matrice de chaînes
 B  =

!8  1  6  !
!         !
!3  5  7  !
!         !
!4  9  2  !

-->delim='\\';
-->B=[B,delim(ones(ma,1))];                                ← rajout d'une colonne
```

```
--> noter que delim(ones(ma,1)) construit un vecteur (ma,1) ou la chaîne delim est répétée sur chaque ligne.
-->B=strcat(B',' & ')          ← transposition et concatenation avec utilisation d'un séparateur
 B  =

 8 & 1 & 6 & \\ & 3 & 5 & 7 & \\ & 4 & 9 & 2 & \\

-->B=strsubst(B,'& \\ &','\\')                                          ← substitutions
 B  =

 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 & \\

-->B=strsubst(B,'& \\','')                        ← B est maintenant une chaîne de caratères
 B  =

 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2

-->delim="c";delim=strcat(delim(ones(na,1)))       ← construction d'un motif répété na fois
 delim  =

 ccc

-->text=['\[ A= \begin{array}{'+delim+'}';
-->       B;
-->       '\end{array} \]' ];                                  ← concatenation matricielle

-->mputl(text,'matrix.tex')                          ← écriture dans le fichier matrix.tex

-->B=string(A);                                          ← une variante pour construire B
-->fmt='%d';  fmt = strcat(fmt(ones(1,na)),' &') + '\n';      ← construction d'un format
-->B= msprintf(fmt,A)             ← construction d'un vecteur de chaîne par écriture formatée
 B  =

 8 &1 &6\n3 &5 &7\n4 &9 &2\n

-->B=string(A);                                    ← une autre variante pour construire B
-->B=strcat(B,' & ','c');                ← concaténation des colonnes de B avec séparateur

-->text= ['\[ A= \begin{array}{'+delim+'}';
-->        B + ' \\' ;        ← B est un vecteur colonne, on concatène chaque ligne avec ' \\'
-->        '\end{array} \]' ]                          ← concaténation matricielle en ligne "+"
 text  =

!\[ A= \begin{array}{ccc}  !
!                          !
!8 & 1 & 6 \\              !
!                          !
!3 & 5 & 7 \\              !
!                          !
!4 & 9 & 2 \\              !
!                          !
!\end{array} \]          !
```

Les chaînes de caractères peuvent aussi être utilisées pour construire des expressions du langage Scilab que l'on peut alors faire évaluer par l'interprète au moyen de la commande `execstr`

```
-->i=3;
-->text = ['function [y]=plus'+string(i)+'(x)';      ← construction d'un vecteur de chaîne
-->'y = x +' + string(i);
-->'endfunction']
 text  =

!function [y]=plus3(x)  !
!                       !
!y = x +3               !
!                       !
!endfunction            !

-->execstr(text);                          ← évaluation par Scilab de la chaîne construite
-->plus3(4)                                ← on dispose maintenant de la fonction plus3
 ans  =

    7.
```

1.4.4 Booléens et matrices de Booléens

Un Booléen peut prendre deux valeurs "vrai" `T` et "faux" `F`. On dispose des deux variables booléennes `%t` et `%f` qui s'évaluent respectivement à `T` et `F` et qui peuvent être utilisée pour construire des matrices de booléen.

L'évaluation des opérateurs de comparaisons (" `==` ", " `>` ", " `>=` ", " `<=` " et "`~=`") produit des résultats de type matrices de booléens. On dispose des opérateurs matriciels termes à termes " `&` " (et) , " `|` " (ou) et "`~`" (not) et des fonctions logiques `and`, `or` et `not` qui prennent en argument une matrice de Booléens.

Les matrices Booléennes sont utilisées dans Scilab avec les expressions conditionnelles et elles servent aussi bien souvent à sélectionner des éléments dans une matrice. Nous avons déjà illustré cette utilisation dans une sections précédente.

```
-->x=-10:0.1:10;
-->y=( (x>=0).* exp(-x) )+ ((x <0).* exp(x));                    ← conversion automa-
tique de booléen vers scalaires.
-->y=bool2s([%t,%f])                                             ← conversion explicite.
 y  =

!   1.    0. !
```

Nous donnons ici un exemple d'utilisation de matrices booléennes. On cherche à tester la véracité d'une expression logique [1] :

```
-->expr= '(x1 & x2 | x3) == ( (x1 & x2 ) | x3)';  ← on cherche à vérifier si l'expression logique est vraie

-->n=size(strindex(expr,'x'),'*')                  ← combien de fois la chaîne 'x' apparaît dans expr. Noter que '*' indique que l'on veut le produit des dimensions d'une matrice.
 n  =

    6.

-->nv=0;
-->for i=1:n
-->  if grep(expr,'x'+string(i))<>[] then  ← est-ce que x1,x2,... apparaissent dans expr ?
-->    nv=nv+1;
-->  end
-->end
-->nv                                              ← nombre de variables xi dans expr
 nv  =

    3.

-->for i=1:nv
-->  expr=strsubst(expr,'x'+string(i),'x(:,'+string(i)+')');
-->end;

-->expr                          ← dans expr on a substitué xi par la chaîne x(:,i) pour i=1:nv
 expr  =

 (x(:,1) & x(:,2) | x(:,3)) == ( (x(:,1) & x(:,2) ) | x(:,3))

-->n=nv ;                                  ← n contient le nombre des xi présents dans expr

-->T=[1;0]
 T  =

!   1. !
!   0. !

-->x=ones(2^n,n);
-->                                        ← tiens et si on utilisait le mystérieux .*. !
-->for i=1:n
-->  x(:,i) = ones(2^(i-1),1).*.(T.*.ones(2^(n-i),1));
-->end
-->x = (x > 0)          ← table donnant toutes les valeurs booléennes possibles pour n variables
 x  =

! T T T !
```

[1] tester si une expression logique contenant des variables booléennes vue comme une fonction Booléenne de ces variables est la fonction constante qui vaut toujours vrai

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

-->execstr('rep=('+expr+')');                    ← toutes les valeurs booléennes possibles pour expr (on construit une expression et on la fait évaluer par l'interprète Scilab)

-->and(rep)                                      ← expr est-elle vraie ?
 ans  =

  T
```

1.4.5 Matrices creuses

Les représentations dites "creuses" des matrices consistent à ne coder que leurs éléments non nuls. En Scilab, le stockage d'une matrice creuses de taille `(m,n)` est effectué de la façon suivante. Un tableau de taille `m` contient des pointeurs vers des descripteurs de lignes. Chaque descripteur de ligne est codé au moyen de deux tableaux, le premier contient les indices des colonnes qui contiennent des éléments non nuls et le deuxième tableau contient les valeurs associées. Certains problèmes qui, décrits sous forme pleine, ne pourraient être stockés en mémoire par Scilab se trouvent accessibles sous forme creuse. Ceci se fait bien évidemment avec une perte d'efficacité en terme de temps de calcul. La fonction de base pour construire des matrices creuses est la fonction `sparse`. Seules les matrices de scalaire et les matrices de booléens disposent de la possibilité d'une représentation creuse dans Scilab.

```
-->A=sprand(100,100,0.1);  ← une matrice creuse (100,100) contenant 10% de valeurs non nulles
-->whos('-type','sparse')                                    ← occupation mémoire
Name                      Type           Size           Bytes

A                         sparse         100 by 100     13360

-->B=full(A);
-->whos('-name','B');     ← occupation mémoire pour la même matrice en représentation pleine
Name                      Type           Size           Bytes

B                         constant       100 by 100     80016

-->timer();inv(B);timer()                                    ← temps de calcul de l'inverse
 ans  =

    0.01

-->timer();inv(A);timer()                                    ← temps de calcul de l'inverse
 ans  =

    0.26
```

1.4.6 Les listes

Les listes Scilab sont construites par les opérateurs `list`, `tlist` et `mlist`. Ces trois opérateurs construisent en fait des structures au sens où ils permettent d'agréger sous un seul nom de variable un ensemble de données de type différents. On notera que ce sont des types récursifs.

- Si on utilise le constructeur `list`, l'accès aux données se fait par un indice. Tout se passe comme si on avait un tableau de données.
- Si on utilise le constructeur `tlist`, l'objet construit a un nouveau type dynamique et l'accès au données peut se faire par l'intermédiaire de noms. Notons aussi que les champs sont dynamiques et qu'on peut rajouter dynamiquement de nouveaux champs. Une `tlist` étant aussi une liste, l'accès aux éléments de la liste par un indice reste possible.
- Le constructeur `mlist` est très voisin du constructeur `tlist`. La seule différence est que l'accès aux éléments de la liste par un indice n'est plus effective (cette opération restant toutefois possible en utilisant les fonctions `getfield` et `setfield`). En revanche on peut surcharger les opérateurs d'extraction et d'insertion pour donner un sens aux accès faits avec un ou plusieurs indices. Les tableaux à plusieurs indices (`hypermat`) sont implémentés dans Scilab au moyen d'une `mlist`.

Voici maintenant, illustrées sur un exemple, les opérations usuelles sur les listes :

```
-->L=list()                                              ← Une liste vide
 L  =

     ()

-->L(2) = testmatrix('magi',3);          ← affectation L(i)=val. Noter que L(1) n'existe pas.
-->L(0) = 34;                                  ← rajout d'un élément au début de la liste
-->L($) = 'X'                                          ← remplace le dernier élément
 L  =

       L(1)

    34.

       L(2)

    Undefined

       L(3)

 X

-->L($+1) = 'Y'                                        ← rajoute un élément à la fin
 L  =

       L(1)

    34.
```

```
      L(2)

   Undefined

      L(3)

 X

      L(4)

 Y

-->[a,b]=L([1,3])                    ← extraction, on peut extraire plusieurs éléments à la fois
 b  =

 X
 a  =

    34.

-->L(2)=null();                                  ← délétion du deuxième élément de la list
-->function w=f(x,y,z); w='arg1:'+string(x)+' arg2:'+ string(y)+' arg3:'+ string(z);endfunction
-->f(L(:))                           ← transformation de la liste en une séquence d'arguments
 ans  =

 arg1:34 arg2:X arg3:Y

-->f(45,L(2:3))
 ans  =

 arg1:45 arg2:X arg3:Y
```

1.5 Programmer avec Scilab

Scilab est aussi un langage de programmation qui permet d'écrire des scripts (suite d'instructions Scilab dans un fichier) et des fonctions (parfois aussi appelées macros). Un fichier contenant une suite d'instructions Scilab et des définitions de fonction est exécuté par Scilab au moyen de la fonction `exec`. On notera que cette fonction admet un argument optionnel qui permet de contrôler le niveau d'affichage à l'écran lors de l'exécution du script. L'usage est de nommer les scripts scilab par des noms de fichier terminés par le suffixe `.sce`. Il est aussi possible au lancement de Scilab d'indiquer un script à exécuter au moyen de l'option `scilab -f <nom-de-script>`. Si le script se termine par l'instruction `quit`, ceci permet de lancer des exécutions de Scilab en mode "batch".

Si un fichier ne contient que des fonctions Scilab, alors l'usage est de le nommer par un nom de fichier au suffixe `.sci`. Ce fichier peut alors être exécuté dans Scilab (ce qui a pour effet de charger les fonctions dans l'environnement courant) par l'instruction `exec` ou par l'instruction `getf`.

On notera aussi que, lors du lancement de Scilab, si un fichier de nom `.scilab` est présent dans le répertoire de base de l'utilisateur (variable d'environnement `HOME`) alors ce fichier de script est éxécuté.

Nous passons maintenant en revue la syntaxe des itérateurs, des branchements et des fonctions en Scilab

1.5.1 Branchement et aiguillages

Ces instructions permettent de conditionner l'exécution d'instructions à la valeur d'une condition. La forme la plus simple est la suivante :

```
if <condition> then <instructions> end
```

Les instructions contenues dans le bloc `<instructions>` ne seront exécutées que si l'évaluation de la condition `<condition>` est `T`. L'évaluation de la condition `<condition>` peut-être une matrice de booléens ou une matrice de scalaire, la condition n'est alors vraie que si tous les éléments de la matrice de booléens sont vrais ou tous les éléments de la matrice scalaire sont non nuls.

A=log(rand(3,3)) if imag(A)==0 then disp('A est une matrice réelle') ; end

Dans la forme à deux branches,

```
if <condition> then <instructions> else <instructions> end
```

la première branche est exécutée quand la valeur de la condition est vraie et la deuxième quand la valeur de la condition est fausse.

```
 A  =

! - 1.5543587  - 1.1076719  - 0.1628187 !
! - 0.2796559  - 0.4073953  - 0.3772698 !
! - 8.4167389  - 0.4645914  - 0.1298622 !

-->if imag(A)==0 then
-->  disp('A est une matrice réélle');
 A est une matrice réélle
-->else
-->   disp('A est une matrice complexe');
-->end
```

Enfin on peut utiliser une forme multibranches ou les `else` succéssifs sont alors introduits par `elseif`. On peut parfois remplacer des instructions conditionnelles successives par une instruction d'aiguillage (`select`), par exemple quand on veut exécuter des instructions en fonction de la valeur d'un nombre ou d'une chaîne de caractères.),

```
select <expr> ,
  case <expr1> then <instructions>
  case <expr2> then <instructions>
  ...
  else <instructions>
end
```

La valeur de `<expr>` est successivement comparée à la valeur de chacune des expressions `<expr1>`, `<expr2>`, ... Dès qu'il y a égalité le bloc d'instructions suivant le `case` est exécute. Si aucun cas d'égalité n'est détecté alors c'est le bloc `else` qui sera (s'il est présent) exécuté.

1.5.2 Itérations

Il y a deux structures de contrôle itératives dans Scilab, la boucle `for` et la structure `while`. Avant de recourir aux itérateurs, il faudra essayer pour des raisons d'efficacité en terme de temps de calcul de voir si les itérateurs ne peuvent pas être remplacés par des instructions vectorielles ou matricielles.

La syntaxe de la boucle `for` est la suivante :

```
for <nom>=<expr>
   <instructions>
end
```

Le bloc d'instructions va être exécuté de façon itérative et à chaque itération la variable `nom` prendra une nouvelle valeur. Si le résultat de l'évaluation de l'instruction `<expr>` est une matrice alors le nombre d'itérations sera fixé au nombre de colonnes de cette matrice et `<nom>` prendra comme valeur les colonnes successives de cette matrice. Si le résultat de l'évaluation de l'instruction `<expr>` est une liste alors c'est la longueur de la liste qui donne le nombre d'itérations et `<nom>` prendra comme valeur les valeurs des éléments successifs de la liste.

Le bloc `<instructions>` peut contenir l'expression `break`. Si au cours des itérations l'instruction `break` est évaluée alors l'exécution de la structure d'itération se termine et l'exécution des instructions reprends à la fin du bloc itératif. La variable `<nom>` garde la valeur qu'elle avait lors de l'exécution du `break`. En l'absence d'instruction `break` la variable `<nom>` n'a plus de valeur à la fin de l'exécution du bloc itératif.

```
-->n=89;
-->for i=2:(n-1)                                  ← itération sur les entiers de 2 à n-1
-->  if pmodulo(n,i)==0 then break;end
-->end

-->if exists('i') then                              ← est-ce qu'un break a eu lieu ?
-->  mprintf('%d divise %d \n',i,n)                  ← si oui i est un diviseur de n
-->else
-->   mprintf('%d est premier\n',n)                        ← si non n est premier
 89 est premier
-->end

-->n=16778;

-->timer();
-->res=[]                                          ← on veut tous les diviseurs de n
 res  =

     []

-->for i=2:(n-1)
-->  if pmodulo(n,i)==0 then
```

```
-->    res=[res,i];                          ← à chaque itération la taille du  vecteur res croît
-->  end
-->end
-->timer()                                   ← temps CPU écoulé depuis l'appel précédent à timer
 ans  =

    0.85

-->res                                                                    ← les diviseurs de n
 res  =

!   2.    8389. !

-->v=2:(n-1);                                          ← calculs vectoriels pour aller plus vite
-->timer();
-->I=find(pmodulo(n,v)==0)                                        ← même calcul mais vectoriel
 I  =

!   1.    8388. !

-->res = v(I)
 res  =

!   2.    8389. !

-->timer()                                          ← comparer au résultat obtenu plus haut !
 ans  =

    0.
```

La syntaxe de l'itérateur `while` est la suivante

```
while <condition>
   <instructions>
end
```

Les instructions contenues dans le bloc `<instructions>` sont exécutées tant que la condition `<condition>` est vraie. L'évaluation de la condition `<condition>` peut être une matrice de booléens ou une matrice de scalaire ; la condition n'est alors vraie que si tous les éléments de la matrice de booléens sont vrais ou tous les éléments de la matrice scalaire sont non nuls.

```
-->while %t                              ← itération infinie, il faut un verb pour quitter le while
-->  val=evstr(x_dialog('Entrez un nombre positif','1.0'));     ← une fenêtre de dialogue
-->  if val==[] then break;end
-->  if size(val,'*') & val >= 0.0 then              ← réponse correcte: on quitte le while
-->    break;
-->  else
-->     x_message('Votre nombre n''est pas positif -->'+string(val))          ← mes-
sage en cas de réponse incorrecte
-->  end
-->end
```

1.5.3 Fonctions Scilab

Définir de nouvelles fonctions dans Scilab se fait au moyen de la construction syntaxique suivante :

```
function [<nom1>,<nom2>,...]=<nom-de-fonction>(<arg1>,<arg2>,...)
   <instructions>
endfunction
```

Dans la définition de la fonction on précise bien sûr le nom de la fonction, la liste de ses arguments d'entrée et la liste des arguments de sortie. On notera qu'une fonction peut renvoyer plusieurs valeurs. L'appel d'une fonction se fait de la façon suivante :

```
<nom-de-fonction>(<expr1>,<expr2>,...)
```

ou encore :

```
[<v1>,<v2>,...<vp>]=<nom-de-fonction>(<expr1>,<expr2>,...)
```

Dans le premier cas la valeur retourné par la fonction est la première valeur de retour et dans le deuxième cas ce sont les `p` premières valeurs de retour qui sont affectées aux variables `<v1>`, `<v2>`,...,`<vp>`. Lors de l'appel de la fonction les expressions fournie en arguments sont d'abord évaluées et leur valeurs sont passées a l'environnement d'exécution de la fonction sous les noms `<arg1>`, `<arg2>`,.... Les arguments des fonctions sont donc passés par valeurs quel que soit leur type. Notons toutefois que, si un argument est un nom de variable, la variable ne sera pas copiée si dans le corps de la fonction la variable correspondante n'est jamais modifiée.

L'exécution d'une fonction se termine normalement quand tout le bloc `<instructions>` a été exécuté, mais elle peut aussi se terminer si l'exécution du corps de la fonction conduit à une instruction `return`. Le retour de la fonction à lieu alors immédiatement après l'instruction `return`. Les variables de retour `<nom1>`,`<nom2>`,... ont alors la valeur courante qu'elles avaient dans l'environnement d'exécution de la fonction.

```
-->function y=f(x); y=2*x;endfunction          ← fonction simple sur une seule ligne. Noter le ; qu'il faut alors ajouter avant la première instruction
-->x=90;
-->f()                                         ← x n'est pas fourni mais il existe dans l'environement courant, l'appel ne provoque pas d'erreur. Mais c'est maladroit !
 ans  =

    180.

-->f(5,7)                                      ← erreur: trop d'arguments fournis
       !--error    58
incorrect number of arguments in function call...
arguments are :
 x

-->[a,b]=f(5)                                  ← erreur: trop d'arguments demandés en retour
           !--error    59
incorrect # of outputs in the function
arguments are :
```

```
 y


-->function y=f(x); z=x; endfunction                    ← nouvelle définition de f
Warning :redefining function: f

-->y=89;
-->z=67;
-->w=f(x)                    ← y n'est pas calculé dans le corps de la fonction f mais il a une valeur dans l'environement d'appel, c'est cette valeur qui est renvoyée
 w  =

    89.

-->z                                                    ← z n'a pas été modifié par l'exécution de f
 z  =

    67.

-->function y=f(); y=x; endfunction
Warning :redefining function: f

-->x=5;
-->y=f()             ← x n'est pas définie localement: c'est la valeur de x dans l'environement appellant qui est utilisée
 y  =

    5.

-->function y=f(); x= 2*x; y=x;  endfunction
Warning :redefining function: f

-->y=f()                                                ← une valeur locale de x est crée
 y  =

    10.

-->x=[56,67];
-->function y=f(); x(1)= 5; y=x  endfunction
Warning :redefining function: f

-->y=f()                                          ← une valeur locale de x est crée et donc x==5
 y  =

    5.

-->// parler de varargin et varargout
-->// parler des arguments nommés

-->function w=f(x,y,z); z=[x,y,z];endfunction
Warning :redefining function: f
```

```
-->f(6,z=78,y=8);                    ← on peut changer l'ordre dans lequel on donne les arguments
```

Comme on l'a vu une fonction admettant n variables en entrée et retournant p valeurs en sortie peut être appelée avec moins d'arguments d'entrée et le nombre de valeurs de retour demandées peut être inférieur à p. Il est possible de contrôler les valeurs utilisées dans un appel de fonction au moyen de la fonction `argn`. . En effet une ligne d'instruction `[lhs,rhs]=argn()` permet d'obtenir dans les variables `lhs` (resp. `rhs`) le nombre de valeurs de retour (resp. d'entrée) utilisées. Cela permet de gérer des arguments optionnels comme illustré sur l'exemple suivant. On notera sur cet exemple que l'on peut utiliser la fonction `error` pour produire une erreur Scilab.

```
-->function [u,v]=f(x,y)
-->  [lhs,rhs]=argn(0)
-->  if rhs <= 0 then error('at least on argument must be given');end
-->  if rhs <= 1 then y=2;end
-->  if lhs == 2 then
-->    u=x; v=y;
-->  else
-->     u=x+y;
-->  end
-->endfunction

-->[u,v]=f(4)
 v  =

    2.
 u  =

    4.

-->function [varargout]=f()
-->  varargout=list(1,2,3)
-->endfunction
Warning :redefining function: f
```

Normalement le nombre d'argument d'entrées doit être inférieur ou égal au nombre d'argument d'entrée de la fonction appelée. Il y a une exception à cela. Si le dernier argument d'une fonction comportant n arguments d'entrées à pour nom `varargin` alors la fonction peut être appelée avec plus de n arguments. Les arguments fournis à partir du n-ième sont stockés dans une liste Scilab de nom `varargin`.

```
-->function [l]=f(x,varargin); l = varargin; endfunction
-->f(0,1,2)                                          ← varagin devient la liste (1,2)
 ans  =
```

```
    ans(1)

 1.

    ans(2)

 2.
```

Une fonction Scilab peut utiliser les variables courantes de l'environnement d'appel mais en lecture seulement. Il est possible de modifier une variable de l'environnement d'appel que pour un ensemble particulier de variables dites globales. Les variables globales doivent être déclarées dans l'environnement initial au moyen de la fonction `global`, et pour être utilisée en écriture elle doivent aussi être déclarées dans les fonctions qui peuvent changer leurs valeurs

```
-->global a;          ← a est maintenant une variable globales, par défaut elle prend la valeur []
-->isglobal(a)                                                          ← On le vérifie ...
 ans  =

  T

-->function f(); global('a'); a=int(10*rand(1,4)); endfunction
-->f()
-->a                                                                    ← L'appel de f a changé a
 a  =

!   2.    7.    0.    3. !

-->
```

1.5.4 Contrôler l'exécution des fonctions

Il est toujours difficile de programmer sans fautes et programmer en Scilab n'échappe pas à cette fatalité. Plusieurs fonctions (ou fonctionalités) permettent d'aider à la détection des fautes et à leur correction. On peut interrompre l'exécution de code Scilab au moyen de la commande `pause` (du menu principal) ou de l'interruption `Ctrl-C`. Le prompt est alors différent pour signaler à l'utilisateur que l'interruption à eu lieu et à quel niveau d'interruption il se trouve. De même que l'exécution d'une fonction Scilab se fait dans un environnement local à la fonction, une pause à pour effet d'interrompre une exécution en cours et de relancer l'exécution de l'interprète dans un nouvel environnement local. Il faut considérer que les environnements d'exécution s'empilent. L'environnement courant est au sommet de la pile. Dans l'environnement au sommet de la pile on a accès (en lecture) aux variables des environnement précédents. l'interruption `Ctrl-C` arrête l'exécution en cours à un endroit non déterminé à l'avance, sachant que l'on peut alors contrôler ou l'on se trouve par la commande `whereami` (ou `where`). Il est cependant parfois préférable d'insérer explicitement la commande `pause` dans une fonction pour être sur de la position du point d'arrêt rajouté.

```
-->a=34;
```

```
-->function y=f(x) ; pause; a = %pi; pause; y=g(x);  endfunction
-->function y=g(x) ; b = %e ;  y=sin(x);pause;  endfunction

-->f(5)
a                    ← pause dans f on peut visualiser les variables de l'environement au ''dessous''
 a  =

    34.

b= 56;                              ← création d'une variable dans l'environement local de la pause

resume                              ← on quitte la pause et on s'arrête à la suivante toujours dans f

a                                                  ← la valeur de a dans l'environement local de f
 a  =

    3.1415927

exists('b')                      ← le b de l'environement local de la pause précédente n'existe plus
 ans  =

    0.

resume                                    ← on quitte la pause et on s'arrête à la suivante dans g

exists('b','local')                              ← b dans l'environement local de la fonction g
 ans  =

    0.

resume                                                                          ← on continue
 ans  =

  - 0.9589243

-->a=g(4);                                                    ← arrêt a la pause dans la fonction g
[y]=resume([1:4]);                  ← sortie de l'environement de la pause avec envoit d'une va-
riable dans l'envireonement de la fonction g

-->a              ← la valeur retournée par g est la valuer de y retournée par la commande resume
 a  =

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

Plutôt que de modifier le code d'une fonction en y insérant des `pause` ce qui nécessite le rechargement de la fonction dans l'environnement courant de Scilab. On peut utiliser la fonction `setbpt` pour fixer un point d'arrêt à une ligne donnée d'une fonction. On continue l'exécution interrompue au moyen de la commande `resume`. On peut aussi interrompre une exécution au moyen des commandes `abort` (resp. `quit`). On revient alors au niveau 0 – le

bas de la pile d'exécution– (resp au niveau en "dessous" du sommet de la pile d'exécution, le sommet étant retiré de la pile).

On peut aussi exécuter pas à pas le contenu d'une fonction au moyen de la commande `exec`. Dans ce cas il n'y a pas d'environnement local d'exécution et il faut donc définir les argument de la fonction avant l'appel de `exec`

```
-->getf('stepf.sci','n');                          ← charger f1 avec getf et l'option 'n'
-->x=int(10*rand(1,4));                                          ← argument de f1
step-by-step mode: enter carriage return to proceed
y=sin(x).*x
 y  =

!   1.8185949    4.5989062    0.    0.4233600 !
y = y+2,
 y  =

!   3.8185949    6.5989062    2.    2.42336 !
return!
```

Noter que par défaut on ne voit pas le source Scilab des instructions si le chargement de la fonction avec `getf` n'a pas été fait avec l'option `'n'`.

1.6 Fonctions d'entrées sorties

Nous avons déjà vu lors des sessions précédentes que si une instruction se termine par une virgule ou un retour à la ligne alors le résultat de son évaluation est affiché dans la fenêtre Scilab. L'affichage n'aura par contre pas lieu si l'instruction est terminée par un point virgule ou si l'instruction fait partie du corps d'une fonction.

Pour afficher explicitement la valeur d'une variable ou d'une expression on peut utiliser les fonctions `disp` ou `print`. Le premier argument de la fonction `print` permet de contrôler où se fera l'affichage. On peut indiquer un nom ou un descripteur de fichier (voir la commande `file`) ou bien utiliser `%io(2)` qui permet l'affichage dans la fenêtre Scilab :

```
-->a=%pi                                          ← le résultat de l'affectation est affiché
 a  =

    3.1415927

-->function y=f(x) ; a = %pi, y = x+a, endfunction
-->f(2),                                              ← l'affichage de a ne se fait plus
 ans  =

    5.1415927

-->function y=g(x) ; a = %pi, print(%io(2),a), y = x+a, endfunction
-->g(2);                                                  ← On force l'affichage de a
 a  =
```

```
3.1415927
```

Il est possible de contrôler le format utilisé pour l'affichage des nombres par les fonctions `disp` et `print` au moyen de la fonction `format`. On notera aussi qu'il est possible de sauver les variables de l'environnement courant au moyen de la commande `save`. On recharge alors les variables dans l'environnement courant avec la commande `load`. Les commandes `save` et `load` utilisent un format binaire machine indépendant. Les sauvegardes se font donc sans pertes de précisions et sont portables.

Pour contrôler plus finement le formatage des entrées sorties plusieurs fonctions sont disponibles dans Scilab. Les fonctions `write` et `read` sont basées sur des formatages `Fortran`. Les descripteurs de fichiers acceptés par ses deux fonctions sont obtenus au moyen de la commande `file`. Les sorties et entrées dans la fenêtre Scilab étant obtenus par les descripteurs `%io(2)` et `%io(1)`.

Nous ne nous étendrons pas plus sur ces fonctions et nous allons plutôt décrire ici les fonctions Scilab qui émulent les fonction d'entrées sorties que l'on utilise en C. Ces fonctions permettent l'écriture et la lecture formatée ou binaire sur des fichiers, sur les sorties et entrées standard de la fenêtre Scilab et sur des chaînes de caractères.

`mprintf`	Écriture sur la sortie standard
`mfprintf`	Écriture dans un fichier
`msprintf`	Écriture dans une matrice de chaînes de caractères
`mscanf`	Lecture sur l'entrée standard
`mfscanf`	Lecture dans un fichier
`msscanf`	Lecture dans une matrice de chaînes de caractères
`fprintfMat`	Écriture formatée d'une matrice dans un fichier
`fscanfMat`	Lecture formatée d'une matrice dans un fichier
`mgetl`	lecture des lignes d'un fichier dans une matrice de chaînes
`mgetl`	lecture des lignes d'un fichier dans une matrice de chaînes
`mputl`	Écriture des lignes d'une matrice de chaînes dans un fichier
`mopen`	ouverture d'un fichier
`mclose`	fermeture d'un fichier

TAB. 1.4 – Quelques fonctions d'entrées sorties

La fonction `mopen` interface la fonction C,`fopen` et sa syntaxe est la suivante :

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

`filename` est une chaîne décrivant un nom de fichier. L'argument `mode` est une chaîne d'au plus trois caractères (dont la valeur par défaut est `'rb'`) qui permet de préciser si le fichier doit être ouvert en lecture (`'r'`), en écriture (`'w'`), en complétion (`'a'`) ou en lecture/écriture (`'r+'`). L'ajout du caractère `'b'` précise que le fichier est un fichier binaire.

Les fonctions d'écriture et de lecture formatées sont proches des fonctions de noms voisins présentes dans la librairie C. La spécification des formats par exemple suit la spécification des formats des fonctions C. L'aspect vectoriel qui est une caractéristique de Scilab est intégré aux fonctions de lecture écriture. Pour l' écriture, les formats sont utilisés sur les lignes des

arguments vectoriels donnés et pour la lecture on peut préciser le nombre de fois où il faut utiliser le format (voire l'utiliser jusqu'à atteindre une fin de fichier). L'exemple suivant illustre cela :

```
-->mprintf('%d',1:4)                          ← le format s'applique aux arguments en explo-
rant les lignes de chaque arguments
1

-->mprintf('%d %d',1:2,3:4)                          ← les arguments en trop sont ignorés
1

2

-->mprintf('%f %f',1)                        ← des arguments manquants provoquent une erreur
1.000000


                  !--error   998
Error: printf: not enough arguments



-->mprintf('%d ',int(10*rand(3,1)))            ← les formats sont réutilisés sur chaque lignes

 3 6 6

-->mprintf('%d\ n',int(10*rand(3,1)))                  ← \ n affiche un retour à la ligne
0


5


6



-->S=msprintf('%d\ n',int(20*rand(5,1)));S';  ← on construit une nouvelle ligne dans la ma-
trice S
 ans  =

!4  17  13  6  18  !

-->A= rand(4,3);
-->fprintfMat('test',A);  B=fscanfMat('test');              ← lecture écriture d'une ma-
trice dans un fichier texte
-->norm(B-A)                                          ← on teste le résultat de la lecture
 ans  =

    8.159D-07

-->fd=mopen('test','r');                              ← ouverture du fichier en lecture
```

```
-->L=mfscanf(-1,fd,"%f%f%f")    ← utilisation d'un même format sur toutes les lignes du fichier
 L  =

!   0.214601     0.5664250    0.5015340 !
!   0.3126420    0.482647     0.4368590 !
!   0.3616360    0.3321720    0.2693120 !
!   0.292227     0.5935090    0.6325740 !

-->norm(L-A)              ← noter évidement que la précision est controllée par le format choisi
 ans  =

    7.989D-07

-->mclose(fd);
```

Nous donnons maintenant un autre exemple de lecture formatée avec des données de type chaîne de caractères et nombres mélangées. On cherche à lire au moyen de la fonction `mfscanf` le fichier `mfscanf.dat` dont le contenu est :

```
// un exemple avec lecture d' un fichier avec separateur de type '[ ]*,[ ]*'
// ---------------------------------------------------------------------------
Agen                          , 47000 , 56
Aigues Juntes                 , 09000 , 78
Aiguilhe                      , 43000 , 78
Ainac                         , 04000 , 56
Ajaccio                       , 20000 , 23
Ajoux                         , 07000 , 34
Albi                          , 81000 , 23
Alencon                       , 61000 , 12


-->fd=mopen('mfscanf.dat','r');
-->mgetl(fd,2);                                    ← je saute les deux premières lignes
-->[n,a,b,c]=mfscanf(-1,fd,'%[^,],%*[, ]%d%*[, ]%d\ n');                    ← lecture

-->n                                               ← nombre d'arguments retournés
 n  =

    3.

-->stripblanks(a)'                                 ← matrice de chaîne (transposée)
 ans  =

!Agen  Aigues Juntes  Aiguilhe  Ainac  Ajaccio  Ajoux  Albi  Alencon  !

-->[b,c]                                           ← données numériques
 ans  =

!   47000.    56. !
!   9000.     78. !
!   43000.    78. !
!   4000.     56. !
!   20000.    23. !
```

```
!   7000.     34. !
!   81000.    23. !
!   61000.    12. !

-->mclose(fd);
```

Pour permettre une écriture binaire qui soit machine indépendante les fonctions Scilab que nous verrons dans la suite écrivent les nombres dans les fichiers au format *little endian*. Ce mode par défaut peut-être supprimé au moyen du paramètre `swap` en lui donnant la valeur `0`. Dans ce dernier cas les lectures écritures se font avec le mode natif de la machine utilisée.

La fonction `mopen` retourne un descripteur de fichier `fd` et éventuellement un indicateur d'erreur.

La fonction `mclose` permet de fermer un fichier qui a été précédemment ouvert par une commande `mopen` et de libérer ainsi le descripteur du fichier ouvert (Le nombre de fichiers que l'on eut ouvrir simultanément est limité). On notera qu'il est possible de fermer tous les fichiers ouverts avec l'instruction `mclose(file())`.

Les fonctions décrites dans la table 1.5 permettent la lecture écriture de données binaires. Les fonctions `mget` et `mput` utilisent le mode de lecture (petit ou grand "endian") qui est précisé lors de l'ouverture d'un fichier avec `mopen`. Mais il est toujours possible de forcer une lecture en petit ou grand "endian" en utilisant un paramètre adéquat.

`mget`	Lecture de données binaires
`mput`	Écriture de données binaires
`mgetstr`	Lecture d'une chaîne de caractères
`mputstr`	Écriture d'une chaîne de caractères
`mtell`	retourne la position courante dans un fichier
`mseek`	déplace la position courante dans un fichier
`meof`	test de fin de fichier

TAB. 1.5 – Quelques fonctions d'entrées sorties binaires

On pourra se reporter aux fonctions Scilab contenues dans le répertoire `SCI/macros/sound` où des fonctions de lecture écriture de fichier de sons constituent d'excellent exemples d'utilisation des fonctions de la table 1.5. Nous donnons simplement ici un exemple sommaire de lecture écriture d'une matrice dans un format binaire.

```
-->x=testmatrix('magic',4);
-->fd=mopen('save.dat','wb');                    ← Ouverture d'un fichier (le 'b' n'est vrai-
ment utile que sous windows)
-->mput(length('x'),'i',fd)                ← Écriture d'un entier. La longueur de la chaîne 'x'
-->mputstr('x',fd)                                          ← Écriture de la chaîne 'x'
 ans  =

    0.

-->mput(size(x,'r'),'i',fd)                        ← Écriture du nombre de lignes (un entier)
-->mput(size(x,'c'),'i',fd)                      ← Écriture du nombre de colonnes (un entier)
```

```
-->mput(x,'d',fd)                                   ← Écriture d'un tableau de doubles
-->mclose(fd)                                       ← Fermeture du fichier
 ans  =

    0.

-->clear x;

-->fd=mopen('save.dat','rb');                       ← Ouverture d'un fichier en lecture
-->l=mget(1,'i',fd)                                 ← lecture d'un entier
 l  =

    1.

-->name=mgetstr(l,fd)          ← lecture d'une chaîne de caractère dont on connaît la longueur l
 name  =

 x

-->m=mget(1,'i',fd)                                 ← lecture des entiers m et n
 m  =

    4.

-->n=mget(1,'i',fd)
 n  =

    4.

-->data=mget(m*n,'d',fd);                           ← lecture de m*n double dans un vecteur
-->code= msprintf('%s=matrix(data,%d,%d);',name,m,n);
-->execstr(code);          ← création de x dans l'environement courant à partir des données lues
-->mclose(fd)                                       ← fermeture du fichier
 ans  =

    0.

-->x                                                ← test
 x  =

!   16.    2.     3.     13. !
!   5.     11.    10.    8.  !
!   9.     7.     6.     12. !
!   4.     14.    15.    1.  !
```

1.7 Graphiques avec Scilab

Les primitives graphique Scilab sont nombreuses et il est hors de propos ici de les décrire complètement et de façon exhaustives. Nous donnerons un aperçu et des règles d'utilisation générales et il faudra se reporter aux manuel en ligne pour une utilisation précise de chaque

fonction.

Nous commençons par un long script Scilab qui fait un petit tour d'horizon de fonctions graphiques. L'exécution de ce script produit la Figure 1.4.

```
-->xbasc();t=linspace(-20*%pi,20*%pi,2000);        ← xbasc efface la fenêtre graphique courante et remet à zéro la liste des ordes graphiques enregistrés
-->param3d1(sin(t),t.*cos(t)/maxi(t),t/100,35,45,'X@Y@Z',[2,4]);  ← courbe dans  ℝ³
-->xs2ps(0,'fig1');                                  ←vers un fichier Postscript

-->xbasc();x=linspace(-%pi,%pi,40); y=linspace(-%pi,%pi,40);
-->plot3d(x,y,sinh(x')*cos(y)) ;                              ← surface dans  ℝ³
-->xs2ps(0,'fig2');

-->xbasc()
-->function [xdot]=derpol(t,x); xdot=[x(2);-x(1)+(1 - x(1)**2)*x(2)];endfunction
-->xf= linspace(-1,1,10);yf= linspace(-1,1,10);
-->fchamp(derpol,0,xf,yf);                          ← champ de vecteurs dans  ℝ²
-->xs2ps(0,'fig3');

-->xbasc(); v=rand(1,2000,'n');
-->histplot([-6:0.4:6],v,[1],'015',' ',[-4,0,4,0.5],[2,2,2,1]);  ← un histogramme
-->function [y]=f2(x); y=exp(-x.*x/2)/sqrt(2*%pi); endfunction;
-->x=-6:0.1:6;x=x';plot2d(x,f2(x),1,"000");   ←une courbe dans  ℝ² qui se superpose
-->xs2ps(0,'fig4');

-->xbasc();polarplot();xs2ps(0,'fig5');   ←une courbe dans  ℝ² en coordonnées polaires

-->xbasc();
-->function [x,y,z]=f3(alp,tet)
-->  r=log(1+abs(alp))
-->  x=cos(alp).*cos(tet);
-->  y=cos(alp).*sin(tet);
-->  z=sinh(alp);
-->endfunction
-->x=linspace(-%pi/2,%pi/2,40);y=linspace(0,2*%pi,20);
-->[x1,y1,z1]=eval3dp(f3,x,y);                          ←construction de facettes
-->plot3d1(x1,y1,z1);         ← surface dans  ℝ³ donnée par une matrice de facettes
-->xs2ps(0,'fig6');

-->xbasc();
-->x=linspace(-%pi,%pi,40); y=linspace(-%pi,%pi,40);
-->contour(x,y,sinh(x')*cos(y),10);                          ← lignes de niveaux
-->xs2ps(0,'fig7');

-->xbasc();plot2d([],[],rect=[0,0,1,1],strf="012");        ←pour fixer une echelle
-->xset('clipgrf');                                ←pour fixer une zone de clipping
-->n=20;
-->rects=[0.8*rand(1,n);0.8*rand(1,n);0.2*ones(1,n);0.2*ones(1,n)];
-->xrects(rects,rand(1,n)*20);                          ←une famille de rectangles
-->xs2ps(0,'fig8');

-->xbasc()
```

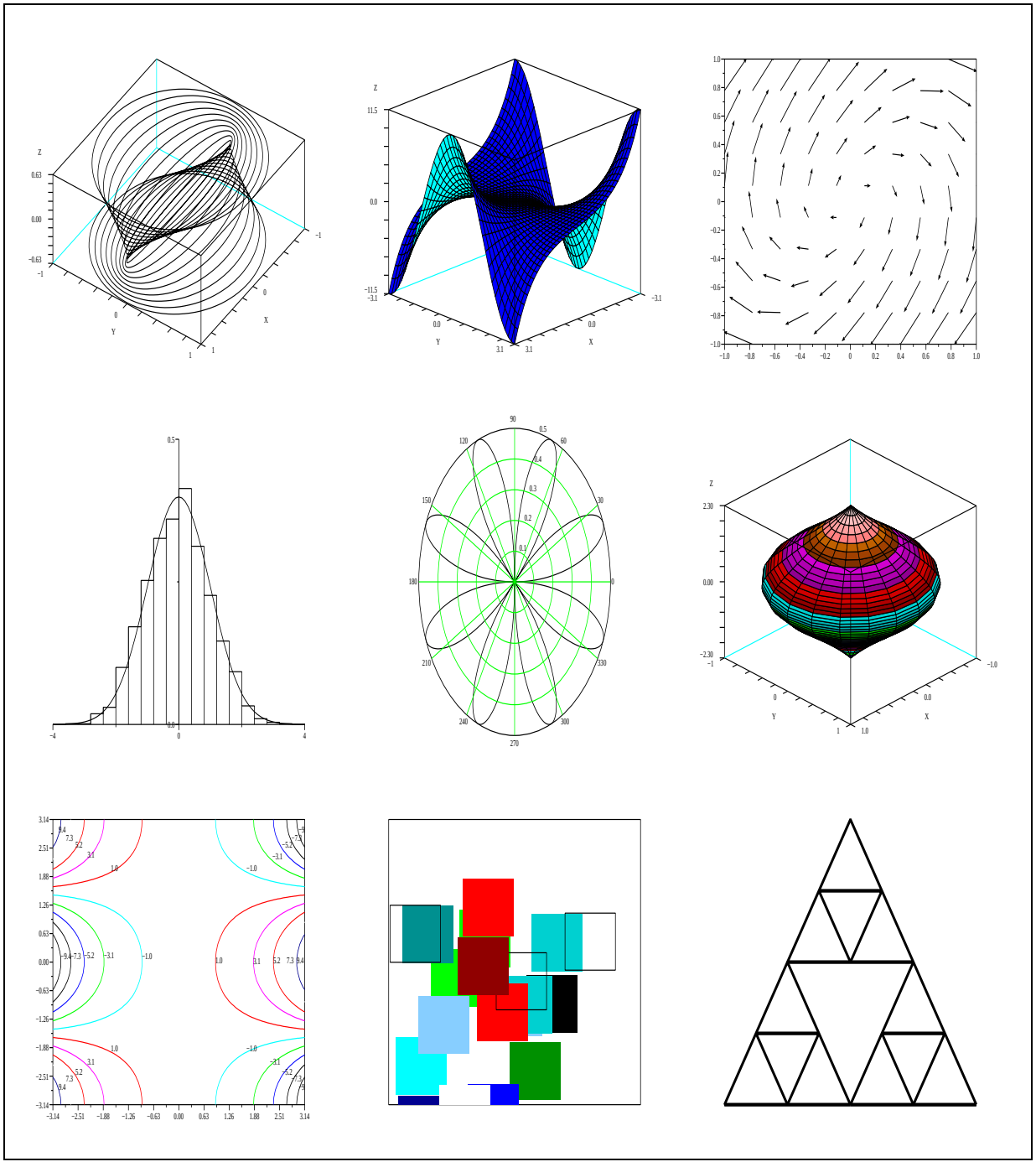

FIG. 1.4 – Petit panorama

```
-->xsetech(frect=[-2*cos(%pi/6),0,2*cos(%pi/6),3])          ←pour fixer une echelle
-->theta=[-%pi/6,%pi/2,%pi*7/6,-%pi/6];
-->T=[cos(theta);sin(theta)];
-->Ts=(1/(1+2*sin(%pi/6)))*rotate(T,%pi);
-->x=[T(1,:);Ts(1,:)]'; y=[T(2,:);Ts(2,:)]';
-->x=[x+cos(%pi/6),x-cos(%pi/6),x];y=[y+sin(%pi/6),y+sin(%pi/6),y+1+2*sin(%pi/6)];
-->xset('thickness',8);                                  ←epaisseur courante du trait
-->xpolys(x,y);                                             ←une famille de polygones
-->xs2ps(0,'fig9');

-->unix(SCI+'/bin/Blatexprs panorama fig[1-9]')          ←appel d'une commande unix
 ans  =

    0.
```

Nous commençons par décrire quelques idées générales sur la façon dont le graphique est implémenté dans Scilab. La sortie graphique courante peut être une fenêtre graphique, un fichier au format Postscript, un fichier au format xfig suivant le "driver" graphique qui est sélectionné (on obtient le driver courant avec la commande `driver()` et on sélectionne un driver au moyen de la même commande avec un argument, par exemple `driver('Pos')` ou avec la commande `xinit` ou bien encore de façon totalement transparente en utilisant des menus de Scilab).

Quand on utilise comme driver une fenêtre graphique Scilab a, par défaut, la particularité d'enregistrer toutes les commandes graphiques dans l'ordre ou elles ont été exécutées. Quand on grandit une fenêtre graphique ou quand on effectue un zoom, Scilab peut donc rejouer les commandes enregistrées et et produire un nouveau graphique adapté au changement de taille. De la même façon on peut rejouer les commandes graphiques après avoir changé de driver pour par exemple obtenir un graphique Postcript. C'est ce qui est fait dans le script précédent au moyen de la commande `xs2ps`.

Le driver courant associé à une fenêtre graphique et qui enregistre les commandes graphiques est le driver `Rec`. On peut rejouer les commandes graphiques au moyen de la commande `xtape` ou de la commande `xbasr`. La commande `xbasc` permet d'effacer la fenêtre courante et d'effacer en même temps les commandes graphiques enregistrées. détruire une fenêtre graphique `xdel` à bien sûr le même effet.

Il peut parfois être utile de supprimer l'enregistrement des commandes graphiques, il suffit pour cela de changer le driver et d'utiliser le driver `X` (ou `W`) à la place de `Rec`.

Par défaut aussi les graphiques apparaissent dans une fenêtre graphique au fur et a mesure de leur exécution. Il est possible de changer ce mode par défaut et de contrôler le moment ou la fenêtre graphique est rafraîchie en changeant le mode pixmap de la fenêtre courante.

Le mode pixmap de la fenêtre courante est un paramètre de la fenêtre et il y en a beaucoup d'autres. Ils définissent le contexte graphique d'une fenêtre. On peut avoir à un moment donné plusieurs fenêtre graphique mais on ne peut changer le contexte graphique que de la fenêtre courante. On obtient le numéro de la fenêtre courante au moyen de `xget('window')` et on change la fenêtre graphique courante par `xset('window',n)`. On obtient les paramètres du contexte graphique de la fenêtre courante au moyen de la fonction `xget` et on les change au moyen de la fonction `xset`.

Mais revenons au mode pixmap. Quand ce mode est sélectionné `xset('pixmap',1)` le graphique n'est plus effectué directement sur la fenêtre graphique mais sur une zone mémoire. La fenêtre graphique n'est rafraîchie que quand la commande `xset('wshow')` est appelée. On peut effacer la zone mémoire utilisée pour le graphique par `wset('wwpc')`. On peut ainsi construire des animations graphiques :

```
-->xsetech(frect=[0,0,10,10])
-->xrect(0,10,10,10)
-->n=100;
-->xset('pixmap',1)
-->driver('X11');
-->for k=-n:n,
-->  a=ones(n,n);
-->  a= 3*tril(a,k)+ 2*a;
-->  a= a + a';
-->  k1= 3*(k+100)/200;
-->  Matplot1(a,[k1,2,k1+7,9])
-->  xset('wshow')
-->  xset('wwpc')
-->end
-->xset('pixmap',0)
```

Un dernier point général est le calcul des échelles graphiques. Les fonctions permettant de dessiner des objets graphiques simples (par exemple des rectangles `xrects`, des polylignes `xpoly`,...) utilisent une échelle graphique courante et il est impératif pour leur utilisation correcte d'avoir d'abord fixé une échelle courante. Cela se fait soit en appelant une fonction de haut niveau, par exemple la fonction `plot2d`, soit au moyen de la fonction `xsetech`.

Les fonctions de haut niveau par défaut fixent une échelle courante de façon a ce quelle soit compatible avec les graphiques précédents. On peut contrôler plus précisément les échelles au moyens de paramètres optionnels.

```
-->xbasc();t=linspace(0,2*%pi);
-->plot2d(t,sin(t))
-->plot2d(t,sinh(t/%pi))        ← l'echelle courante s'adapte pour contenir les deux courbes (Figure 1.5 gauche)
-->[a,b,c,d]=xgetech();
-->b                                                ← l'echelle courante: [xmin,ymin,xmax,ymax]
 b  =

!   0.  - 1.    7.    4. !

-->xbasc();
-->plot2d(t,sin(t))
-->plot2d(t,sinh(t/%pi),strf="000")      ← sans changer l'echelle courante (Figure 1.5 droite)
-->[a,b,c,d]=xgetech();
-->b                                                ← l'echelle courante: [xmin,ymin,xmax,ymax]
 b  =

!   0.  - 1.    7.    1. !
```

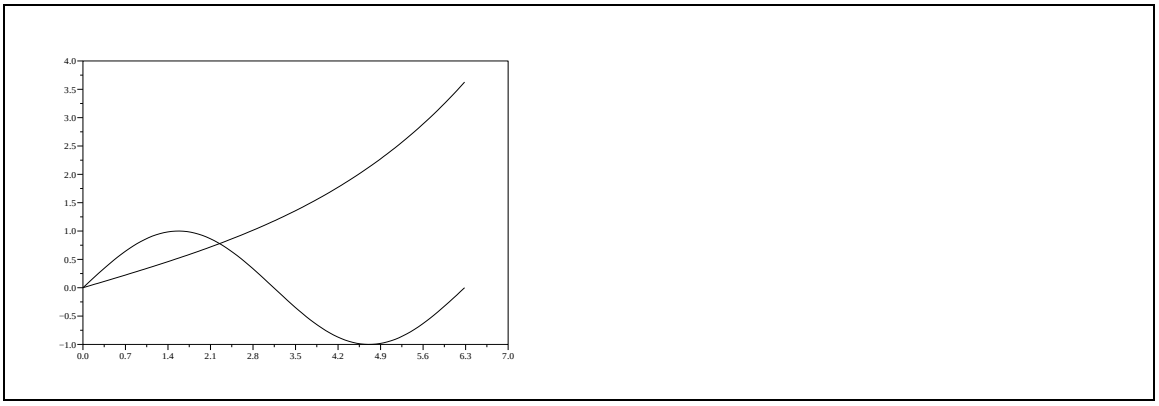

FIG. 1.5 – Utilisation des echelles

1.8 Interfaçage

1.8.1 Écriture d'une interface

Supposons qu'on dispose d'une librairie de fonctions externes écrites en C. La librairie contient par exemple la fonction `geom` qui effectue `m*n` tirages aléatoires selon une loi géométrique de paramètre `p` et stocke les résultats dans un tableau d'entiers `y` de taille `m` x `n`. Savoir ce que fait exactement `geom` n'est pas vraiment utile pour nous ici. Disons simplement que, étant donnés les entiers `m` et `n` et le paramètre `p`, cette fonction retourne le tableau `y`. Voilà le code "C" de la procédure `geom` (qui est fort simple), la seule chose importante ici étant sa liste d'appel :

```
#include <stdlib.h>

int geom(int m,int n,double p,int y[])
{
  int i;
  if ( p >= 1.0 )
    {
      cerro("p doit-etre < 1 "); return 1;
    }
  for (i = 0 ; i < m*n ; i++)
    {
      double z = drand48();
      y[i] = 1;
      while ( z < 1.0-p ) { z = drand48(); y[i] ++; }
    }
  return 0;
}
```

On souhaite donc rajouter dans Scilab une fonction (une "primitive") de même nom `geom` dont la syntaxe d'appel est `y=geom(m,n,p)`. La fonction `geom` est supposée effectuer `m*n` tirages de loi géométrique et renvoyer le résultat sous la forme d'une matrice Scilab de taille `m` x `n`.

Pour ce faire, il faut écrire une interface, c'est-à-dire un programme (disons à nouveau en C) qui va être chargé de faire les conversions et le passage des données entre l'interprète

champ,champ1,fchamp	champs de vecteurs dans le plan
contour,fcontour,contour2d, contourf, fcontour2d	courbes de niveau
errbar	rajoute des barres d'erreurs à une courbe 2D
eval3d,eval3dp,genfac3d	utilitaires pour construire des facettes pour un graphique 3D
fec	dessin 2D d'une fonction définie sur une triangulation
geom3d	calcul de coordonnées
grayplot,fgrayplot	courbes de niveau
graypolarplot	courbes de niveau sur courbe polaire
hist3d	histogramme 3D
histplot	histogramme 2D
isoview	change les échelles graphiques pour obtenir une représentation isométrique
legends	rajoute des légendes sur un graphique 2D
Matplot,Matplot1	dessin 2D des valeurs d'une matrice
param3d,param3d1	Courbes paramétrique en 3D
paramfplot2d	animation de courbes 2D paramétrées
plot	courbes 2D
plot2d,fplot2d ,plot2dxx	Courbes 2D
plot3d,plot3d1,fplot3d, fplot3d1,plot3d2,plot3d3	dessin 3D d'une surface
polarplot	Courbes en coordonnées polaires
Sfgrayplot, Sgrayplot	courbes de niveau
subplot	subdivise une fenêtre graphique en sous fenêtre

TAB. 1.6 – Fonctions graphiques

Scilab et la fonction C `geom`. Nous appellerons `intgeom` l'interface pour la fonction `geom` (il faut écrire une interface pour chaque fonction que l'on veut interfacer).

Il existe dans Scilab diverses façons d'écrire une interface. Nous allons utiliser ici la méthode qui nous semble la plus simple. De nombreux exemples d'écriture d'interfaces sont fournis dans le répertoire

`SCI/examples`.

Le répertoire `SCI/examples/interface-tutorial-so/` donne aussi un exemple très simple d'interface, destinée aux utilisateurs débutants, proche de celle décrite ici.

Pour écrire cette interface particulière, on a utilisé des fonctions de la librairie d'interfaçage (`CheckRhs`, `CheckLhs`, `GetRhsVar` et `CreateVar`), des variables de la librairie d'interfaçage (`LhsVar`) et des fonctions internes Scilab (`cerro`, aussi utilisée dans `geom`). Pour utiliser les fonctions et les variables précédentes, il faut rajouter le fichier d'en-tête `stack-c.h` dans le fichier `intgeom.c`.

L'interface de la fonction `geom` s'écrit alors de la façon suivante :

```
#include "stack-c.h"

extern int geom(int m, int n, double p, int y[]);

int intgeom(char *fname)
{
```

xarc,xarcs	dessine une (ou plusieurs) ellipse ou une partie d'ellipse
xarrows	dessine un ensemble de flèches
xaxis	dessine un axe gradué
xclea	efface une zone rectangulaire
xclip	fixe une zone de clip
xfarc,xfarcs	peint une (ou plusieurs) ellipses
xfpoly,xfpolys	peint le contenu d'une (ou plusieurs) polyligne
xfrect	peint le contenu d'un rectangle
xgrid	rajoute une grille sur un graphique
xnumb	rajoute des nombres sur un graphique
xpoly,xpolys	dessine une (ou plusieurs) polyligne
xrect	dessine un rectangle
xrects	dessine ou peint un ensemble de rectangles
xrpoly	dessine un polygone régulier
xsegs	dessine un ensemble de segments non connectés
xstring,xstringb	dessine une chaîne de caractères
xstringl	coordonnées d'un rectangle entourant le dessin d'une chaîne de caractères
xtitle	rajoute un titre

TAB. 1.7 – Primitives Graphiques

```
int l1, m1, n1, l2, m2, n2, m3, n3, l3;
int minlhs=1, maxlhs=1, minrhs=3, maxrhs=3;
int m,n,y;  double p;

/* 1 - Vérification du nombre d'arguments d'entrée et de sortie */
CheckRhs(minrhs,maxrhs) ;
CheckLhs(minlhs,maxlhs) ;

/* 2 - Vérification du type des arguments et retour des pointeurs */
GetRhsVar(1, "i", &m1, &n1, &l1);
GetRhsVar(2, "i", &m2, &n2, &l2);
GetRhsVar(3, "d", &m3, &n3, &l3);

if ( m1*n1 != 1 || m2*n2 != 1 || m3*n3 != 1)
  {
    cerro("Erreur: les arguments de geom doivent etre scalaires");
    return 0;
  }
/* 3 - Extraction des valeurs */
m = *istk(l1); n = *istk(l2) ; p = *stk(l3);

/* 4 - Création de la variable de retour */
CreateVar(4,"i",&m,&n,&y);
if ( geom(m,n,p,istk(y)) == 1 ) return 0;

/* 5 - Spécification de la variable de retour */
LhsVar(1) = 4;
return 0;
```

`driver`	choix d'un driver graphique
`graycolormap`, `hotcolormap`	tables de couleurs
`xbasc`	efface la fenêtre graphique courante et les ordres graphiques enregistrés
`xbasimp`	imprime la fenêtre courante
`xbasr`	redessine le contenu d'une fenêtre graphique
`xchange`	changement de coordonnées
`xclear`	efface la fenêtre courante
`xclick`	attente d'un "click" souris
`xdel`	détruit une fenêtre graphique
`xend`	termine une session graphique
`xget`	donne des valeurs du contexte graphique
`xgetech`	donne l'échelle de la fenêtre courante
`xgetmouse`	position et évènement souris
`xgraduate`,`graduate`	utilitaire pour la graduation des axes
`xinfo`	rajoute une information dans la zone message d'une fenêtre graphique
`xinit`	initialise un driver graphique
`xlfont`	charge des fonts
`xload`	recharge un graphique sauvé par `xsave`
`xname`	change le nom de la fenêtre graphique courante
`xpause`	pause
`xs2fig`,`xs2gif`,`xs2ppm`,`xs2ps`	convertit le graphique courant au format `Xfig`, `gif`, `ppm` ou `Postscript`
`xsave`	sauve un graphique dans un format rechargeable avec `xload`
`xselect`	sélectionne une fenêtre et la met au premier plan
`xset`	fixe des valeurs du contexte graphique
`xsetech`	fixe l'échelle courante
`xtape`	gestion de l'enregistrement des ordres graphiques
`xtitle`	rajoute un titre
`winsid`	retourne la liste des fenêtre graphiques

TAB. 1.8 – Primitives liées aux fenêtres graphiques

```
}
```

A première vue, cette fonction peut paraître un peu compliquée. En fait on ne va pas taper tout ce code, mais plutôt partir d'un exemple tout fait qu'on va adapter à notre cas. Puisque tous les programmes d'interface sont bâtis sur le même moule, les modifications à faire sont très faciles à réaliser avec quelques retouches à l'éditeur. L'interface précédente pourrait par exemple être utilisée presque telle quelle pour n'importe quelle fonction C qui aurait une liste d'appel semblable à celle de `geom`.

Comment cette interface marche-t-elle ? Quand sous l'interprète Scilab, on tape la commande `y=geom(m,n,p)`, les arguments sont évalués et leurs valeurs sont stockées dans un tableau (que l'on appellera pile d'appel) dans l'ordre où ils apparaissent dans la liste d'appel. D'autre part le nombre d'arguments de retour attendus (ici `y`) est connu. La fonction d'interfaçage `intgeom` doit tout d'abord contrôler que le nombre d'arguments transmis et attendus au retour sont corrects. Cela est fait en utilisant `CheckLhs` et `CheckRhs` ; si le nombre d'arguments ne correspond pas aux bornes spécifiées ces fonctions génèrent une erreur et provoquent la sortie de l'interface.

Dans l'interface, chaque variable Scilab est repérée par un numéro, d'abord les variables d'entrée puis les variables de sortie. Ici par exemple, la fonction Scilab étant `y=geom(m,n,p)`, les variables d'entrée `m,n,p` ont les numéros 1,2,3 et la variable de sortie `y` a le numéro 4.

Ensuite, il faut vérifier que chaque élément présent dans la liste d'appel a le bon type (est-ce une matrice, une chaîne de caractères, une fonction ?) et la bonne taille. Enfin il faut récupérer un pointeur vers les données pour pouvoir les transmettre à la fonction interfacée ; c'est ce qui est fait dans la deuxième partie de la procédure `intgeom`.

La commande suivante :

```
GetRhsVar(2, "i", &m2, &n2, &l2);
```

a pour effet de vérifier que le deuxième argument de la liste d'appel est bien de type numérique (`"i"`) et de renvoyer dans `m2` et `n2` les dimensions de la matrice et dans `l2` une adresse pour accéder aux données (une conversion des données en entiers est faite). Si le type ne convient pas une erreur est générée.

Le deuxième argument étant de type entier on obtient un pointeur d'entiers avec `istk(l2)`. Le deuxième argument de `geom` qui doit être un entier est donc finalement récupéré dans l'interface par `n= *istk(l2)`.

On notera qu'il est prudent de vérifier que `m2*n2` vaut bien 1 car une utilisation de `n = *istk(l2)` dans un appel `y=geom(10,[],2.5)` donnerait n'importe quoi.

Le troisième argument est de type double (`"d"`) et on y accède par `p=*stk(l3)`. De manière générale, disposant des dimensions des arguments, on doit effectuer des vérifications et en cas d'erreur on peut utiliser la fonction `cerro` pour afficher un message puis faire un `return` (Scilab prend alors le relais). On notera d'ailleurs que dans la fonction `geom`, on a aussi utilisé la fonction `cerro` pour imprimer un message d'erreur.

Avant d'appeler la fonction `geom`, il faut créer la variable numéro 4, soit `y`, et réserver de la place pour la stocker. La variable `y` doit être une matrice d'entiers de taille `m`×`n` et la fonction `geom` veut un pointeur d'entiers. La commande : `CreateVar(4,"i",&m,&n,&y)` de la quatrième partie de `intgeom` se charge de créer dans la pile d'appel un nouvel objet Scilab avec le numéro `4` (une matrice `m`×`n`) et à nouveau on obtient une adresse pour accéder aux données `y` (les données sont entières et sont donc accessibles via `istk(y)`)

La syntaxe d'appel de `CreateVar` est la même que celle de `GetRhsVar` sauf que les 4 premiers arguments sont des entrées et le dernier une sortie calculée par `CreateVar`. Après l'appel de `geom`, il ne reste plus qu'à renvoyer à Scilab le résultat ce que réalise la cinquième partie de la procédure `intgeom`. Le nombre de variables attendues par Scilab a déjà été contrôlé au début de l'interface (`CheckRhs`) et il ne reste plus qu'à indiquer par l'intermédiaire de `LhsVar(i)=j` les positions des variables que l'on veut renvoyer : l'instruction `LhsVar(1)=4` signifie "la première variable de sortie est la variable numéro 4". C'est Scilab qui contrôle alors tout seul d'éventuelles conversions de données à effectuer.

Il n'est bien sûr pas question de décrire ici toutes les fonctions de la bibliothèque d'interfaçage. Le lecteur intéressé pourra se reporter au répertoire

`SCI/examples/interface-tour-so`

pour une description au travers d'exemples de toutes les possibilités de la librairie. La connaissance des 4 fonctions décrites ici, (`CheckLhs`, `CheckRhs`, `GetRhsVar`, `CreateVar`) permet d'interfacer la plupart des fonctions C.

Dans le répertoire `SCI/examples/interface-tour-so` on montre en particulier comment interfacer une fonction C qui a elle-même une fonction comme paramètre d'entrée, ou comment on peut appeler l'interprète Scilab à l'intérieur d'une interface. C'est évidemment un peu plus compliqué, mais des exemples simples sont donnés pour réaliser ce type d'interface.

Les utilisateurs familiers des `mexfiles` Matlab pourront se reporter au répertoire

`SCI/examples/mex-examples`

pour constater que leurs fichiers `mex` (librairie d'interfaçage de Matlab) marchent quasiment tels quels dans Scilab.

1.8.2 Chargement et utilisation

Il nous reste une deuxième étape importante : comment charger et utiliser le code précédent dans Scilab. Nous allons suivre ici la méthode qui est utilisée dans le répertoire

`SCI/examples/interface-tour-so`.

Pour que l'exemple soit plus réaliste nous allons interfacer deux fonctions, la fonction `geom` que nous avons vu précédemment et la fonction système `srand48` qui permet d'initialiser le générateur aléatoire `drand48`. La fonction système `srand48` ne renvoie pas de valeur et admet pour argument un entier, voila son interface :

```
int intsrand48(char *fname)
{
  int l1, m1, n1;
  CheckRhs(1,1);
  CheckLhs(0,1);
  GetRhsVar(1, "d", &m1, &n1, &l1);
  if ( m1*n1 != 1 ) {
      cerro("Erreur: srand48 attend un argument scalaire");
      return 0;
  }
  srand48((long int) *stk(l1));
  LhsVar(1) = 0; /* pas de valeur renvoyée */
  return 0;
}
```

Nous avons donc deux fonctions à interfacer et on supposera que les codes C correspondants sont écrits dans un unique fichier `intgeom.c` (contient `intgeom`, `geom` et `intsrand48`) . On notera qu'il n'y a en fait aucune contrainte sur la répartition des fonctions dans les fichiers. Il faut maintenant charger ces codes dans Scilab pour définir deux nouvelles primitives `geom` et `srand48`.

Pour cela, il faut compiler le code, le charger dynamiquement (en chargeant une librairie partagée `.so` sous Unix ou `.dll` sous Windows) et indiquer les nom Scilab (`geom` et `srand48`) que l'on veut donner aux fonctions dont les codes sont interfacés par `intgeom` et `intsrand48`.

Tout cela va pouvoir s'effectuer depuis Scilab d'une façon indépendante du système hôte (Unix, Windows), des compilateurs disponibles, etc, de la façon suivante. Au moyen d'un éditeur de texte on va constituer un fichier de nom `builder.sce`. Ce nom est canonique et un utilisateur qui rencontre un fichier de nom `builder.sce` sait qu'il faut exécuter un tel fichier pour configurer une contribution, un interface, etc ....

```
// This is the builder.sce
// must be run from this directory

ilib_name  = 'libealea' // interface library name

// objects files

files = ['intgeom.o'];


libs  = []  // other libs needed for linking

// table of (scilab_name,interface-name)
// for fortran coded interface use 'C2F(name)'

table =['geom', 'intgeom';
        'srand48','intsrand48'];

// do not modify below
// ------------------------------------------
ilib_build(ilib_name,table,files,libs)
```

Une base de départ pour construire le fichier `builder.sce` est le fichier d'exemple `SCI/examples/interface-tutorial-so/builder.sce` que l'on peut facilement adapter à un nouvel interfaçage.

Dans ce fichier on trouve les informations suivantes :

- `ilib_name` contient une chaîne de caractères, c'est le nom que l'on veut donner la "librairie" (un ensemble d'interfaces) que l'on est en train de construire.
- `files` est un vecteur de chaînes de caractères donnant la liste des fichiers objets qui constituent la librairie. Ici on trouve juste le fichier `intgeom.o`. Noter que les fichiers objets sont notés avec un suffixe `.o` et cela même si l'on se trouve sous Windows. Un même fichier `builder.sce` doit pouvoir s'exécuter sans changements sous Windows ou sous Unix.
- `libs` est un de chaînes de caractères donnant une liste de librairies nécessaires à la création de la librairie partagée.
- `table` est une matrice de chaînes de caractères ou on donne la correspondance entre un nom de fonction Scilab et un nom d'interface. On trouve par exemple sur la première ligne `'geom', 'intgeom';` qui indique que la fonction Scilab `geom` est interfacée au moyen de l'interface `intgeom`

Il suffit alors de faire exécuter ce fichier par Scilab (il est impératif pour se faire que le répertoire courant de Scilab soit le répertoire ou se trouve le fichier `builder.sce`) pour créer

une librairie partagée et créer un fichier `loader.sce` qui permettra de charger la librairie dans Scilab.

```
-->exec builder.sce
-->ilib_name  = 'libalea';      // interface library name
-->files = ['intgeom.o']        // objects files
-->libs  = []                   // other libs needed for linking
-->table = [ 'geom', 'intgeom'; // table of (scilab_name,interface-name)
-->          'srand48','intsrand48'];
-->// do not modify below
-->// -----------------------------------------------
-->ilib_build(ilib_name,table,files,libs);
   generate a gateway file
   generate a loader file
   generate a Makefile: Makelib
   running the makefile
```

Après l'exécution de ce fichier, de nouveaux fichiers sont crées dans le répertoire courant.

Le plus important d'entre eux est le fichier `loader.sce`. Il permet le chargement dans Scilab de la nouvelle librairie au moyen de la fonction `addinter`. En pratique on exécute donc une fois `builder.sce` pour compiler la librairie puis à chaque nouvelle session Scilab on exécute `loader.sce` pour charger la librairie (noter que le chargement peut être effectué de n'importe où, il n'est pas nécessaire de se trouver dans le répertoire de `loader.sce` pour l'exécuter).

```
// appel du loader qui est dans un répertoire distant
-->exec SCI/contrib/geom/loader.sce

-->// generated by builder.sce: Please do not edit this file
-->// -----------------------------------------------------
-->libalea_path=get_file_path('loader.sce');
-->functions=[ 'geom';
-->            'srand48';
-->];
-->addinter(libalea_path+'/libalea.so','libalea',functions);
Loading shared executable addinter-linux-so//libalea.so
shared archive loaded
Linking libalea (in fact libalea_)
Interface 0 libalea
```

Citons pour information les autres fichiers générés. On trouve un fichier `libalea.c` appelé "gateway". Il contient une procédure appelée "procédure de gestion des interfaces". Elle sert d'aiguillage pour gérer la table de correspondance `table`.

On trouve également un fichier `Makelib` qui est un fichier `Makefile` qui dépend du système hôte et de l'environnement de compilation (Unix, Windows/Visual C++, Windows/Absoft). Il permet la création de la librairie partagée.

Il ne reste plus qu'à tester cette nouvelle primitive `geom` avec le petit script qui suit. On compare graphiquement (Figure 1.6) l'histogramme empirique obtenu par simulation et celui donnée par la théorie

```
-->n=10000; pr=0.2 ;
-->y=geom(1,n,pr);   // appel de la nouvelle primitive
```

```
-->N=20;  i=0:N;
// tests des résultats

-->z=[]; for i1=i, z=[z,size(find(y==i1),"*")];end
-->plot2d3("onn",i',z'/n,[1,3],"161","Simulation");

-->zt=0;for i1=1:N; zt=[zt,pr*(1-pr)^(i1-1)];end
-->plot2d1("onn",i',zt',[-2,6],"100","Theorie");
```

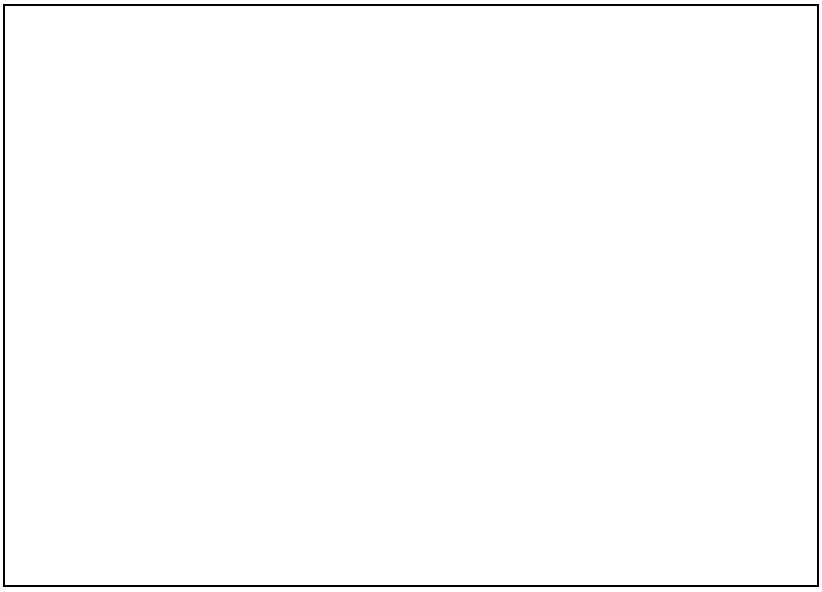

FIG. 1.6 – Histogramme des données

1.8.3 intersci

Il existe dans Scilab un utilitaire appelé `intersci` qui permet de générer un fichier d'interface à partir d'un fichier de description de la fonction à interfacer. `intersci` ne permet pas toute la souplesse d'une interface écrite à la main en C mais peut être utilisé pour certaines interfaces simples ou comme base de départ. Dans une version récente `intersci` génère du code compatible avec celui que nous avons décrit dans les paragraphes précédent.

La documentation complète de `intersci` se trouve dans le répertoire `doc` de Scilab et de nombreux exemples sont décrits dans le répertoire

`SCI/examples/intersci-examples-so`.

Nous nous limitons ici à un exemple simple, interfacer la fonction `ext1c` :

```
#include "machine.h"

int F2C(ext1c)(n, a, b, c)
     int *n;
     double *a, *b, *c;
```

```
{
  int k;
  for (k = 0; k < *n; ++k)
      c[k] = a[k] + b[k];
  return(0);
}
```

On notera que cette fonction n'a que des arguments de type pointeurs et que le nom de la fonction est encapsulé dans un appel de macros `C2F(ext1c)` (c'est une limitation de `intersci` qui doit interfacer à la fois des fonctions C et Fortran). Pour interfacer cette fonction il faut écrire un fichier de description disons `ex01fi.desc` (le suffixe `.desc` est imposé) :

```
ext1c a b
a  vector m
b  vector m
c vector  m

ext1c   m a b c
m integer
a double
b double
c double

out sequence c
***********************
```

Le texte écrit dans le fichier indique que l'on va créer une fonction Scilab `[c]=ext1c(a,b)` où `a`,`b` et `c` sont trois vecteurs de même taille. La première ligne décrit les arguments d'entrée et la dernière les arguments de sortie. La fonction `ext1c` pour s'exécuter doit appeler la fonction C ou Fortran `C2F(ext1c)` avec quatre arguments : un pointeur d'entier qui contiendra la taille des vecteurs et trois pointeurs de "double" qui permettrons d'accéder aux valeurs contenues dans les vecteurs Scilab `a`,`b` et `c`. La place nécessaire pour stocker `c` est réservée dans l'interface avant l'appel de la fonction C `C2F(ext1c)`.

En exécutant `intersci` au moyen de la commande :

```
bin/intersci-n ex01fi
```

on va générer l'interface `ex01fi.c` et un fichier de type "builder" (comme au paragraphe 1.8.2) `ex01fi_builder.sce`.

```
#include "stack-c.h"
/*****************************************
 * SCILAB function : ext1c, fin = 1
 *****************************************/

int intsext1c(fname)
   char *fname;
{
 int m1,n1,l1,mn1,m2,n2,l2,mn2,un=1,mn3,l3;
 CheckRhs(2,2);
 CheckLhs(1,1);
 /*  checking variable a */
```

```
 GetRhsVar(1,"d",&m1,&n1,&l1);
 CheckVector(1,m1,n1);
 mn1=m1*n1;
 /*  checking variable b */
 GetRhsVar(2,"d",&m2,&n2,&l2);
 CheckVector(2,m2,n2);
 mn2=m2*n2;
 /* cross variable size checking */
 CheckDimProp(1,2,m1*n1 != m2*n2);
 CreateVar(3,"d",(un=1,&un),(mn3=n1,&mn3),&l3);
 C2F(ext1c)(&mn1,stk(l1),stk(l2),stk(l3));
 LhsVar(1)= 3;
 return 0;
}
```

On se reportera au répertoire
`SCI/examples/intersci-examples-so`
où le même exemple est repris de façon plus complète. L'ensemble des opérations à effectuer étant récapitulées dans le fichier `ex01.sce` :

```
// appel de intersci-n au moyen d'un Makefile
G_make('ex01fi.c','ex01fi.c');
// execution du builder cr\'ee par intersci-n
files = ['ex01fi.o' , 'ex01c.o'];
libs  = [] ;
exec('ex01fi_builder.sce');
// execution du loader
exec loader.sce
// test de la fonction
a=[1,2,3];b=[4,5,6];
c=ext1c(a,b);
if norm(c-(a+b)) > %eps then pause,end
```

1.8.4 Utilisation de la commande link

Pour terminer, il convient de parler d'un cas (encore) plus simple où on peut charger des fonctions dynamiquement dans Scilab sans avoir à écrire une interface. On utilise pour cela les fonctions Scilab `link` et `call`. Une contrainte est imposée, on doit modifier la fonction C pour que sa liste d'appel ne contienne que des pointeurs.

```
#include <stdlib.h>
int geom1(int *m, int *n, double *p, int y[])
{
  int i;
  if ( *p >= 1.0 )
    {
      cerro("p doit-etre < 1 "); return 1;
    }
  for (i = 0 ; i < (*m) * (*n) ; i++)
    {
```

```
      double z = drand48();
      y[i] = 1;
      while ( z < 1.0 - *p ) { z = drand48(); y[i] ++; }
    }
  return 0;
}
```

Comme précédemment il faut créer une librairie partagée et la charger cette fois dans Scilab avec la commande `link`. Il existe donc deux façons de charger une librairie partagée dans Scilab mais qui correspondent à deux usages différents : (`addinter` rajoute des primitives Scilab et nécessite l'écriture d'interfaces alors que scifunlink ne fait que rajouter des fonctions dans la table des fonctions accessibles par les primitives non linéaires `ode`, `optim`,... et la primitive `call`.

```
-->names=['geom1'] // les points d'entrées à rajouter
                   // ici un seul.
-->files = 'geom1.o' // les fichiers objets.
-->flag = 'c'
-->ilib_for_link(names,files,[],"c");
   generate a loader file
   generate a Makefile: Makelib
   running the makefile
```

Dans ce qui précède `flag='c'` permet de préciser que la fonction `geom1` est écrite en C. *Cette précision est nécessaire car sur certains systèmes le compilateur Fortran ajoute un "blanc souligné" (underscore) au début ou à la fin des noms des points d'entrée ce que ne fait pas le C.*

On charge alors dynamiquement cette fonction dans Scilab en exécutant le fichier `loader.sce`.

```
-->exec loader.sce  // chargement de la librairie
-->link(geom1_path+'/libgeom1.so',['geom1'],'c');
Loading shared executable ./libgeom1.so
shared archive loaded
Linking geom1 (in fact geom1)
Link done
```

On peut alors appeler la fonction C `geom1` depuis Scilab et obtenir sa sortie comme une variable Scilab. On doit envoyer à `geom1` les paramètres d'entrée `m,n` et `p` et récupérer la matrice de sortie `y`. C'est la commande `call` qui permet de faire cet appel. La syntaxe est un peu longue car on doit fournir les valeurs des variables d'entrée, mais aussi leur type C et leur position dans la liste d'appel de `geom1`. C'est en effet la fonction `call` qui doit faire le travail de conversion que nous avions précédemment codé dans un interface. Cela donne par exemple :

```
-->m=3;n=4;p=0.3;

-->y=call("geom1",m,1,"i",n,2,"i",p,3,"d","out",[m,n],4,"i");
```

Dans cette instruction, on indique en premier argument le nom de la fonction C appelée (c'est aussi le point d'entrée passé à `link`). Les arguments suivants, jusqu'à la chaîne `out`,

donnent l'information relative aux variables d'entrée. Les arguments situés après la chaîne `out` donnent, pour leur part, l'information relative aux variables de sortie.

À chaque variable est associé un triplet de paramètres. On interprète par exemple les trois paramètres `m,1,"i"` comme : passer la valeur `m` comme premier argument de `geom` de type `int`. En sortie, on aura : `y` (premier argument à gauche du signe égal) est une matrice de dimension `[m,n]`, quatrième paramètre de `geom`, de type `int`.

Évidemment, on peut encapsuler l'instruction `call(...)` dans une fonction Scilab `y=geom(m,n,p)` et en profiter pour faire des tests (par exemple de dimensions) sur les arguments passés à notre fonction C.

Une autre utilisation de la commande link se fait dans le contexte suivant : certaines primitives Scilab comme par exemple les fonctions `ode`, `fsolve` ou `optim` admettent des arguments de type fonction que l'on appellera fonctions externes. Par exemple pour chercher les zéros de la fonction $\cos(x) * x^2 - 1$, on peut construire une fonction Scilab :

```
function y=f(x)
y=cos(x)*x^2-1
```

et utiliser ensuite cette fonction `f` comme argument de la fonction `fsolve` :

```
-->y0=10; y=fsolve(y0,f)
 y  =

    11.003833
-->f(y)
 ans  =

    3.819D-14
```

On peut parfois avoir envie, pour des raisons d'efficacité, de coder ce genre de fonctions dans un langage compilé (C, C++ ou Fortran). Dans ce cas, il ne faut pas vraiment écrire une interface, car la fonction `fsolve` a déjà été prévue pour fonctionner avec des fonctions `f` externes. `fsolve` nous impose par contre une contrainte sur la liste d'appel de la fonction `f` qu'elle peut reconnaître (voir le help de `fsolve`). L'exemple précédent pourrait être codé en C sous la forme :

```
#include <math.h>
void f(int *n,double *x,double *fval,int *iflag)
{
  *fval = cos(*x)*(*x)*(*x) - 1;
}
```

On procède comme précédemment pour compiler et charger la fonction C `f` dans Scilab :

```
-->ilib_for_link('f','f.o',[],'c');
   generate a loader file
   generate a Makefile: Makelib
   running the makefile

-->exec loader.sce
```

et on indique à `fsolve` que le problème à résoudre concerne la fonction `f` écrite en C en lui passant en argument la chaîne `"f"` :

```
-->y0=10; y=fsolve(y0,"f")
 y  =

    11.003833
```

Cela est valable pour la plupart des primitives Scilab admettant des argument fonctionnels.

Deuxième partie

Examples illustratifs

Chapitre 2

Programmer : Arbres et matrices de ramification

Le but de ce T.P de programmation Scilab est la synthèse d'images d'arbres par les matrices de ramification [3] [1].

Les arbres que l'on va générer seront des arbres binaires : chaque arête mère donne naissance à exactement deux arêtes filles (ou aucune pour les arêtes qui conduisent aux feuilles de l'arbre). Un exemple d'arbre binaire est donné dans la figure 2.1

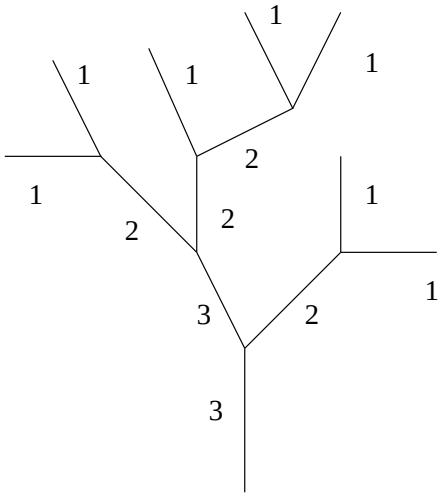

FIG. 2.1 – Un exemple d'arbre et les ordres des arêtes

Sur cet arbre binaire on peut définir l'ordre d'une arête ou d'un arbre (c'est l'ordre de l'arête issue de la racine de l'arbre) qui se calcule récursivement de la façon suivante : soit t un arbre, si l'arbre est réduit à une arête son ordre $od(t) = 1$ sinon on considère les deux sous arbres de l'arbre t, soient $\mathrm{R}(t)$ et $\mathrm{L}(t)$ et on a alors la formule :

$$od(t) \stackrel{\text{def}}{=} \begin{cases} \max(od(\mathrm{R}(t)), od(\mathrm{L}(t))) & \text{si} \quad od(\mathrm{R}(t)) \neq od(\mathrm{L}(t)) \\ od(\mathrm{L}(t)) + 1 & \text{si} \quad od(\mathrm{R}(t)) = od(\mathrm{L}(t)) \end{cases} \tag{2.1}$$

On voit ainsi que partant des arêtes d'ordre 1 (celles qui on un noeud feuille) on peut, en

utilisant la formule précédente, calculer l'ordre de toute les arêtes de l'arbre (un exemple est donné sur la figure (2.1).

On appelle biordre d'un noeud le couple (i, j) (avec $i < j$)) des ordres des arêtes filles issues du noeud considéré.

Si l'ordre d'une arête est k alors les biordres possibles sont

$$(1,k),(2,k),\ldots,(k-1,k) \quad \text{et} \quad (k-1,k-1)$$

Il y a donc k biordres possible pour une arête d'ordre k.

Pour générer aléatoirement une structure d'arbre, on part de la racine de l'arbre d'ordre fixé N. Puis récursivement on génère par tirages aléatoires des biordres successifs pour construire les arêtes filles.

Pour tirer un biordre au hasard, afin de générer les descendants d'une arête d'ordre k, il faut se donner une loi discrète sur $\{1,\ldots,k\}$notée P_k.

Pour générer un arbre d'ordre $od(t) = N$, il faut donc se donner $N-1$ lois de probabilité discrètes $(P_k)_{k=2,N}$. La loi P_k étant une loi discrète sur$\{1,\ldots,k\}$. Pour simplifier on rajoute la loi P_1 qui ne sera pas utilisée puisqu'une feuille n'a pas de descendants. Cela revient à se donner une matrice triangulaire inférieure (dite *matrice de ramification*) P de taille $N \times N$ vérifiant $0 \leq p_{ij} \leq 1$ et $\sum_{j=1}^{N} p_{ij} = 1$. La loi P_k se lit sur la k-ième ligne de la matrice P.

2.0.5 Matrices de ramifications

On commence par écrire une fonction qui étant donné un ordre N et un paramètre de choix renvoie une matrice de ramification. Pour commencer la fonction doit renvoyer deux types de matrice, soit une matrice identité `N` × `N`, soit une matrice `N` × `N` de la forme :

$$\begin{pmatrix} 1 & & & \\ \alpha^1 & \alpha^1 & & \\ \alpha^1 & \alpha^2 & \alpha^2 & \\ \alpha^1 & \alpha^2 & \alpha^3 & \alpha^3 \\ \ldots & & & \end{pmatrix} \quad \text{avec} \quad \alpha \stackrel{\text{def}}{=} \frac{1}{2} \tag{2.2}$$

```
-->function [N,mat]=matrice_ramification(N,val)
-->  select val                                  ← Selection du type de matrice à construire
-->   case 1 then mat=eye(N,N);                  ← Matrice Identité
-->   case 2 then
-->    mat= (1/2).^[1:N-1];                     ← Puissance termes à termes pour construire ( α^1& ... &α^(N-1) )
-->    mat=ones(N-1,1)*mat  ;                   ← Copier N fois la même ligne
-->    mat=tril(mat);                           ← Extraire la partie triangulaire inférieure
-->    d=diag(diag(mat),1);d($,:)=[];           ← La diagonale principale
-->    mat = [mat,zeros(N-1,1)]+d;
-->    mat = [eye(1,N);mat];                    ← Rajout de la première ligne
-->  end
-->endfunction

-->[N,mat]=matrice_ramification(3,2); mat
 mat  =

!   1.     0.     0.   !
```

```
!   0.5    0.5     0.   !
!   0.5    0.25    0.25 !
```

2.0.6 Tirage aléatoire des ordres des arêtes filles

On écrit maintenant une fonction `calcul_ordre` qui étant donné un ordre `k` utilise un matrice de ramification `mat` pour faire un tirage aléatoire des ordres des deux arêtes filles issues de l'arête d'ordre `k`. On utilise la `k`-ième ligne de la matrice de ramification pour effectuer un tirage dans $\{1,\ldots,k\}$. Si la valeur `k` est tirée les deux arêtes filles on pour ordre `k-1` sinon la valeur tirée donne l'ordre de l'une des arêtes filles et l'autre à alors pour ordre la valeur `k`. Le choix d'attribution entre fille gauche et droite se fait alors avec une loi uniforme.

```
-->function [ordreg,ordred]=calcul_ordre(k)
-->  o1=k;
-->  o2=grand(1,'markov',mat(k,1:k),1)                  ← loi sur 1,...,k en utilisant mat
-->  if o2==k then
-->    o1=k-1; o2= k-1;                                ← les deux arêtes filles sont d'ordre k-1
-->  end
-->  ale= grand(1,'uin',1,2) ;                                ← 1 ou 2 avec loi uniforme
-->  if ale==1,                                        ← Attribution des ordres gauche et droit
-->    ordreg=o1; ordred=o2;
-->  else
-->    ordreg=o2; ordred=o1;
-->  end
-->endfunction
```

2.0.7 Construction d'un arbre

On utilise ici les listes Scilab pour coder un arbre binaire de la façon suivante :

- Une feuille sera codée par la liste `T=list(1)` (`1` est l'ordre d'une feuille)
- Un arbre t d'ordre `k` sera codé par `T=list(k,Tg,Td)` où `Tg` et `Td` codent respectivement les sous arbres gauches et droits de l'arbre t.

On écrit maintenant une fonction récursive Scilab qui construit une liste codant un arbre aléatoire obtenu à partir d'une matrice de ramification `mat`

```
-->function [L]=construire(k)
-->  if k==1 then
-->    L=list(1);                                              ← Liste associée à une feuille
-->  else
-->    [ordreg,ordred]=calcul_ordre(k);                ← Tirage des ordres des arêtes filles
-->    L=list(k,construire(ordreg),construire(ordred));          ← On suit la définition
-->  end
-->endfunction

-->N=2; [N,mat]=matrice_ramification(N,1);                       ← Matrice identité 2x2
-->L=construire(N)                                                      ← Un exemple
 L  =
```

```
   L(1)

2.

   L(2)

    L(2)(1)

1.

   L(3)

    L(3)(1)

1.
```

2.0.8 Utilisation des graphes Scilab pour visualiser l'arbre

Le codage précédent des arbres n'est pas très parlant, la sortie écran de Scilab devenant rapidement inadapté pour visualiser un arbre, même de petite taille. Avant d'utiliser directement le graphique Scilab, nous allons construire un graphe Scilab à partir de notre arbre et directement utiliser les fonctions de représentation de graphes de Scilab pour la visualisation. Pour ce faire il faut pouvoir attribuer des positions dans le plan aux noeuds de notre arbre. Nous allons utiliser un algorithme du a D.Knuth dont l'idée est la suivante. Au cours de l'algorithme de construction de l'arbre on va conserver pour chaque noeud sa profondeur et on va attribuer a chaque noeud un numéro. Les numéros attribués aux noeuds correspondent à une numérotation obtenue en faisant un parcourt en profondeur d'abord de l'arbre. On change donc le codage précédent en rajoutant la profondeur et une numérotation dans les listes en remplaçant le scalaire `ordre` par une matrice `[ordre,numero,profondeur]`.

On modifie la fonction `construire` en conséquence. Elle admet maintenant trois arguments, l'ordre et la profondeur du noeud courant et le premier numéro attribuable à un noeud de l'arbre à générer. En retour la fonction `construire` renvoie un arbre sous forme de liste Scilab et le denier numéro attribué à un noeud dans la construction de l'arbre.

La nouvelle version de `construire` prends maintenant la forme suivante :

```
-->function [L,nmax]=construire(k,numero,prof)
-->  if k==1 then
-->    L=list([1,numero,prof]);                                    ← Une feuille
-->    nmax=numero;                          ← Le dernier numéro utilisé dans le cas feuille
-->    return;
-->  end
-->  [ordreg,ordred]=calcul_ordre(k);
-->  [Lg,nmg]=construire(ordreg,numero,prof+1)                     ← Le sous-
arbre gauche utilise numero comme premier numéro à attribuer
-->  [Ld,nmd]=construire(ordred,nmg+2,prof+1);          ← Le sous arbre droit utilise le dernier numéro attribué a l'arbre gauche plus 2 comme premier numéro
-->  L=list([k,nmg+1,prof],Lg,Ld);       ← On rajoute la racine d'ordre k et de numéro nmg+1
-->  nmax=nmd;
-->endfunction
```

Il suffit maintenant d'attribuer aux noeuds de l'arbre une abscisse proportionnelle aux numéros des noeuds et une ordonnée proportionnelle à la profondeur des noeuds pour obtenir une représentation visuelle de l'arbre.

Nous commençons par construire deux matrices :

- Une dont le nombre de colonnes est le nombre d'arêtes de l'arbre et chaque colonne contient le numéro du noeud père et le numéro du noeud fils de l'arête.
- Une autre dont le nombre de colonnes est le nombre de noeuds de l'arbre et chaque colonne contient l'ordre, le numéro et la profondeur du noeud correspondant

On construit ces deux matrices simultanément à l'aide d'une fonction récursive utilisant `L` le résultat de la fonction `construire` :

```
-->function [M,pos]=construire_matrice(L,M,pos)
-->  if size(L)== 3 then                                    ← Ce n'est pas une feuille !
-->    M=[M,[L(1)(2);L(2)(1)(2)],[L(1)(2);L(3)(1)(2)]];     ← Deux arêtes à rajouter
-->    [M,pos]=construire_matrice(L(2),M,pos);              ← Passer au sous arbre gauche
-->    [M,pos]=construire_matrice(L(3),M,pos);              ← Passer au sous arbre droit
-->  end
-->  pos=[pos,L(1)']                          ← Rajouter les informations du noeud courant
-->endfunction
```

On notera que les matrices `M` et `pos` sont à la fois donnée en entrée et en sortie de la fonction `construire_matrice`. Les arguments des fonctions étant passés par valeur, on ne peut pas les modifier. Par contre on peut en modifier une copie et renvoyer cette copie dans les arguments de sortie de la fonction.

Il ne reste plus qu'à tester le code. On note que si l'on choisit une matrice Identité `N`×`N` pour matrice de ramification, on obtient un arbre régulier de hauteur `N` et l'ordre des arête diminue strictement à chaque division. On trouvera sur la Figure 2.2 un arbre obtenue en utilisant la matrice identité avec $N = 4$ (à gauche) et un exemple avec la matrice 2.2 à nouveau avec $N = 4$ (à droite).

```
-->N=4;val=1;
-->[N,mat]=matrice_ramification(N,val);
-->[L,nmax]=construire(N,1,1);
-->[g,rect]= construire_graphe(L,nmax);

-->rep=[2 1 1 2 1 2 2 2 2 2 2 2 2];
-->plot_graph(g,rep,rect);                                  ← Voir Figure 2.2
```

2.1 Représentation graphique directe

On cherche maintenant à obtenir une représentation graphique des arbres générés en utilisant les primitives graphiques de Scilab. Pour obtenir une représentation graphique, on va construire des segments qui représentent les arêtes de l'arbre et dessiner à l'écran l'ensemble des segments à l'aide de la fonction `xsegs`. On dispose de plusieurs degrés de liberté :

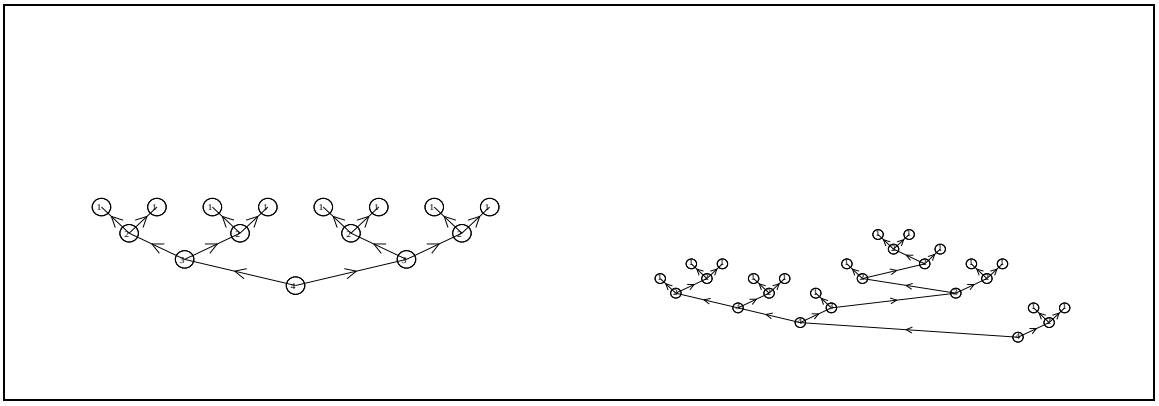

FIG. 2.2 – Représentation de l'arbre

- On peut faire dépendre la couleur et l'épaisseur du trait utilisé de l'ordre du segment à tracer.
- A chaque ramification, les angles de rotation θ_1 et θ_2 des branches filles par rapport à la direction de la branche mère peuvent dépendre aussi de l'ordre de la branche mère.
- Quand une branche mère et une branche fille ont même ordre, on peut imaginer qu'il s'agit d'une branche principale de l'arbre et donc diminuer l'angle de rotation entre mère et fille dans ce cas la.

Nous allons à nouveau construire l'arbre par une fonction récursive. Au cours de la récursion nous voulons construire une matrice qui codera l'ensemble des segments et leurs ordres. Pour des arbres de grande taille procéder comme dans la fonction `construire_matrice` sera très pénalisant à cause des recopies et réallocation des matrices passées à la fois en argument d'entrée et de sortie. Nous allons donc plutôt ici utiliser une variable globale ou plutôt une fonction `empiler(x)` qui reçoit en argument un vecteur ligne et qui le stocke dans une matrice globale.

Si la variable globale s'appelle `M` empiler doit en théorie juste effectuer l'opération `M=[M;x]`. Cette opération qui va être répétée un grand nombre de fois n'est pas vraiment efficace car elle s'accompagne d'une copie de `M`. Pour en augmenter la rapidité `M` sera allouée par blocs. On utilisera une `tlist` pour coder la pile avec les champs suivants :

- `val` est une matrice `pxq` où `q` est la taille des vecteurs à stocker dans la pile.
- `pos` est l'indice de la ligne courante que l'on peut utiliser pour stocker un vecteur dans la pile.
- `libres` est le nombre de lignes encore libres dans la pile.
- `inc` est le nombre de lignes à rajouter dans `val` quand la pile est pleine. C'est donc la taille d'allocation des blocs.

On écrit maintenant la fonction `empiler` et la fonction `valeurs` qui donne la matrice contenant toutes les valeurs empilées.

```
-->function []=empiler(x)                          ← Rajoute un vecteur ligne au som-
met de la pile, la taille de x doit être fixe
-->  global %pile;
-->  n=length(x);buf_size=1000;                    ← buf_size nous donne la taille des blcos
-->  if typeof(%pile)=='constant'  then             ← Creation de la pile
```

```
-->    %pile=tlist(['pile','val','pos','libres','inc'],0*ones(buf_size,n),1,buf_size,buf_size);
-->  end
-->  if %pile.libres > 1 then
-->    %pile.val(%pile.pos ,:)=x;                                          ← On stocke x
-->    %pile.pos = %pile.pos + 1;                          ← On incrémente la position libre
-->    %pile.libres = %pile.libres - 1;       ← On décrémente le nombre de positions libre
-->  else
-->    %pile.val=[%pile.val;0*ones(%pile.inc,n)];       ← On rajoute un block dans la pile
-->    %pile.libres = %pile.libres + %pile.inc;       ← On mets à jours le nombre de posi-
tions libres
-->    empiler(x);                                                         ← On stocke x
-->  end
-->endfunction

-->function m=valeurs()               ← On extrait de la pile la matrice des valeurs stockées
-->  global %pile;
-->  m=%pile.val(1:%pile.pos-1,:);        ← La dernière valeur stockée est sur la ligne  pos-1
-->endfunction
```

La fonction `calculer` construit un arbre et stocke au fur et à mesure les segments qui le compose dans la pile.

```
-->function []=calculer(cp,cd,k)        ← calcul les arêtes fille du segment cd ↦ cp d'ordre k
-->  empiler([cd(1),cp(1),cd(2),cp(2),k]);       ← On empile les informations sur le segment
-->  if k==1,return;end                                          ← Arrêt sur une feuille
-->  [ordreg,ordred]=calcul_ordre(k);            ← Calcul des ordres des deux branches filles
-->  l1=l(ordreg),l2=l(ordred);                        ← Les longueurs des branches filles
-->  delta1=del(ordreg)*%pi/180,delta2=del(ordred)*%pi/180;      ← les angles de rotation
-->  if ordreg==k then
-->    delta1=delta1/5;                                ← la branche principale tourne moins
-->  end
-->  if ordred==k then
-->    delta2=delta2/5;                                ← la branche principale tourne moins
-->  end
-->  [cp1,cp2]=newp(cp,cd)                             ← calcul des deux nouveaux noeuds
-->  calculer(cp1,cp,ordreg);                ← calcul du nouveau segment qui suit cp ↦ cd1
-->  calculer(cp2,cp,ordred);                ← calcul du nouveau segment qui suit cp ↦ cd2
-->endfunction

-->function [cp1,cp2]=newp(cp,cd)                        ← Rotations pour trouver les coor-
données des deux noeud fils
-->  dir=cp-cd;dir = dir/norm(dir,2)
-->  diro=[dir(2),-dir(1)]
-->  cp1= cp + l1*cos(delta1)*dir + l1*sin(delta1)*diro
-->  cp2= cp + l2*cos(delta2)*dir - l2*sin(delta2)*diro
-->endfunction
```

Et il ne reste plus qu'a tester notre code :

```
-->clearglobal %pile;
```

```
-->N=10;val=1;
-->[n,mat]=matrice_ramification(N,val);
-->l=[1:N]/N;   // les longueurs a utiliser pour les aretes de nombre i
-->del=20*ones(1,N); // l'angle a utiliser pour les aretes de nombre i

-->xbasc();
-->nc=N;cmap=[((1:nc).^2)/nc;1:nc;((1:nc).^2)/nc]'/nc;
-->xset("colormap",cmap) // l'angle a utiliser pour les aretes de nombre i

-->calculer([0,0],[0,-1],N);

-->m=valeurs();
-->x=m(:,1:2);
-->y=m(:,3:4);
-->plot2d(0,0,1,"010"," ",[mini(x),mini(y),maxi(x),maxi(y)])
-->for i=1:N
-->  I=find(m(:,5)==i);
-->  xset("thickness",i)   // l'epaisseur du trait dépend de l'ordre de l'arete.
-->  xsegs(x(I,:)',y(I,:)',N-i);
-->end
```

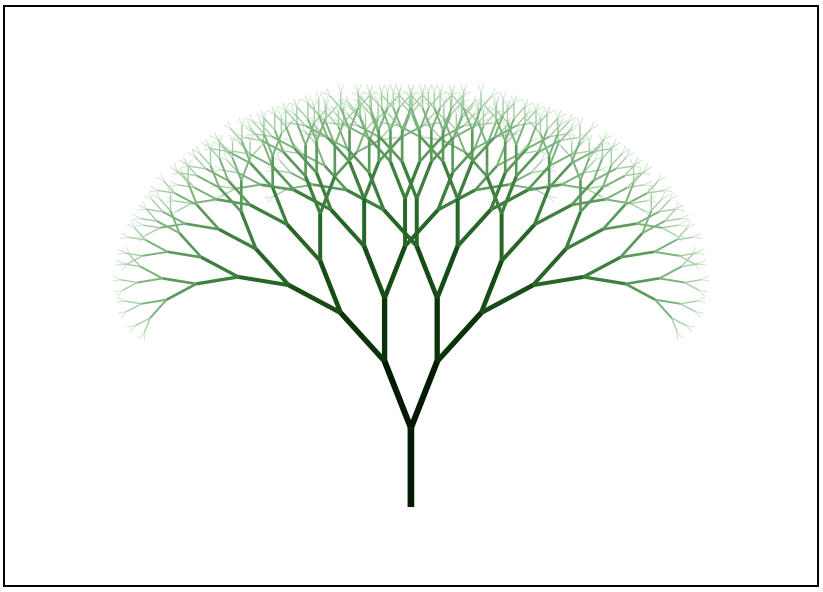

FIG. 2.3 – Arbre 1

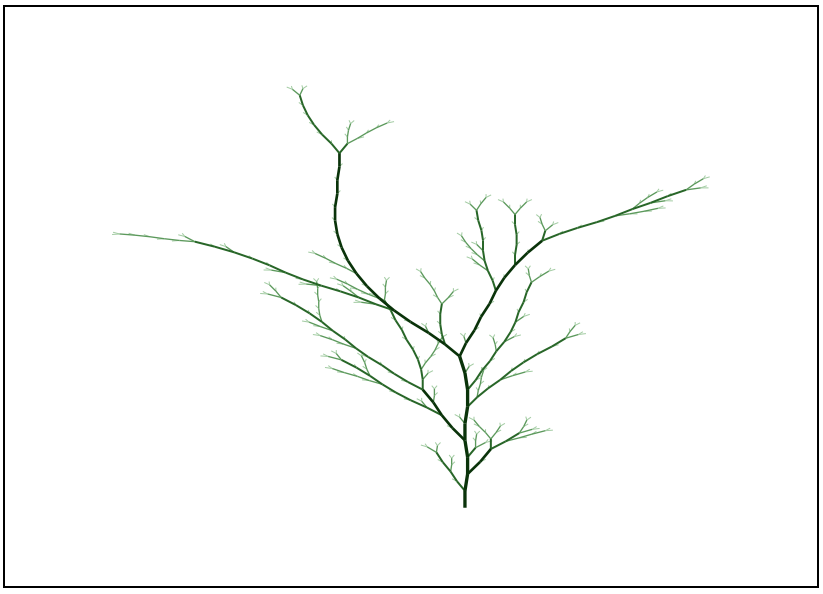

FIG. 2.4 – Arbre 2

Chapitre 3

Programmer : Arbres et DOL-systèmes

3.0.1 DOL-systèmes

Le but de ce T.P de programmation Scilab est la synthèse d'images d'arbres. Les arbres sont obtenus aux moyens d'expression grammaticales et de règles de ré-écriture [2].

On génére tout d'abord des chaînes de caractères au moyen de règles de production ou de ré-écriture, puis on reinterprete ces chaînes de caractères en les explorant sequentiellement et en associant une action à chaque lettre lue.

Soit $\mathcal{A}$ un alphabet, on désignera par $\mathcal{A}^*$ l'ensemble des mots que l'on peut construire avec cet alphabet (y compris le mot vide) et par $\mathcal{A}^+$ les mots non vide de $\mathcal{A}^*$.

On appellera DOL la donnée d'un triplet $(\mathcal{A}, \omega, P)$ ou $\mathcal{A}$ est un alphabet, ω est un mot non vide ($\omega \in \mathcal{A}^+$) appelé axiome et P est un ensemble de règle de réécriture ($P \subset \mathcal{A} \times \mathcal{A}^*$). Les règle de réécriture sont donc des couples $(a, \xi) \in \mathcal{A} \times \mathcal{A}^*$ que l'on écrit en général sous la forme $a \to \xi$.

A chaque lettre de l'alphabet $\mathcal{A}$ doit correspondre au moins une règle de réécriture. Si aucune règle de réécriture n'est spécifiée explicitement pour $a \in \mathcal{A}$ alors on supposera que la règle $a \to a$ est valide.

Enfin on appellera un DOL-système la donnée d'un L-système vérifiant la propriété suivante : pour tout $a \in \mathcal{A}$ il n'existe qu'un $\xi \in \mathcal{A}^*$ tel que $a \to \xi$. Le D veut dire déterministe : pour réécrire un élément de $\mathcal{A}$ il n'y a qu'une solution.

Comment utilise-t-on un DOL-système ? on part de l'axiome ω et on remplace toutes les lettres de ω par le mot associé dans la règle de réécriture. On obtient ainsi un nouveau mot ω_1 et on peut repeter le processus un nombre finies de fois.

Considerons l'exemple suivant :

$$\begin{aligned} \mathcal{A} &= \{F, H, +, -\} && (3.1)\\ H &\to F + H + F && (3.2)\\ F &\to H - F - H && (3.3)\\ \omega &= F && (3.4) \end{aligned}$$

partant de l'axiome ω on obtient après une première application des règles de réécriture :

$$\omega_1 = H - F - H$$

puis à l'étape 2 :

$$\omega_2 = F + H + F - H - F - H - F + h + F$$

et ainsi de suite

On codera un DOL-sytème en Scilab au moyen d'une fonction qui reçoit en entrée une chaine de caratère et qui en sortie renvoit la chaîne de caratère obtenue après application des ré-écritures.

```
-->function [w]=Irewrite0(v)
-->  w=strsubst(v,'F','X+F+X'); ← Règle : F ↦ H+F+H. On protège les H nommés X pour l'instant.
-->  w=strsubst(w,'H','F-H-F');                                    ← Règle : H ↦ F-H-F.
-->  w=strsubst(w,'X','H');                                        ← On renomme les X en H.
-->endfunction
```

Pour effectuer N étape de réécriture il suffit d'effectuer une boucle. On écrit une fonction chargée de cela :

```
-->function [w]=rewrite(macro,init,n)
-->  w=init;                                                       ← init est l'axiome.
-->  for i=1:n, w=macro(w);end                                ← init On applique les ré-écriture
-->endfunction
```

Et on obtient :

```
-->w=rewrite(Irewrite0,'F',3)                          ← 3 niveaux de ré-écriture à partir de F
 w  =

 H+F+H-F-H-F-H+F+H+F-H-F+H+F+H+F-H-F+H+F+H-F-H-F-H+F+H
```

3.0.2 Interprétation des chaînes

Après N niveaux de réécriture dans un DOL-système on obtient une chaîne ω_n. On va interpréter maintenant cette chaîne pour en faire un dessin. L'idée est la suivante, on passe en revue les caractère de la chaîne ω_n du premier au dernier et on associe à chaque caractère de $\mathcal{A}$ une action graphique.

On reprend l'exemple précédent avec $\mathcal{A} = \{H, F, +, -\}$. On se place dans le plan et on se donne un point courant x et une direction d :

- Quand on lit un symbole H ou un symbole F (et de façon plus générale quand on lit une lettre), on se déplace dans la direction d d'une longueur L. Le nouveau point courant devient $x + l * d$.
- Quand on lit un $+$ (resp $-$) on change la direction d d'un angle δ (resp $-\delta$).

L et δ sont deux constantes que l'on se donne et que l'on pourra faire varier.

En parcourant la chaîne ω_n, on va décrire une courbe dans le plan. La fonction qui suit `calcul_segments` calcule les différent segments qui vont composer la courbe et renvoie ces segments dans deux matrices `x` et `y`.

```
-->function [xs,ys]=calcul_segments(w,delta,l)
-->  delta = delta*%pi/180;
-->  Rm = [cos(delta),-sin(delta);sin(delta),cos(delta)]
-->  Rp = [1,-1;-1,1].*Rm;
-->  L=length(w);
-->  res=ones(4,L);                          ← Matrice pour stocker les segments calculés
-->  count=1;                                 ← premier indice utilisable pour stocker un segment
-->  pt=[0;0];dir=[0;1];                                  ← Point et direction initiaux.
-->  for i=1:L
-->    select part(w,i)                                  ← Le $i$-ième caractère de w.
-->     case '+' then
-->      dir= Rp*dir;
-->     case '-' then
-->      dir= Rm*dir;
-->    else
-->      newpt=pt + l*dir;
-->      res(:,count)=[pt(1);newpt(1);pt(2);newpt(2)];
-->      pt = newpt;
-->      count=count+1;
-->    end
-->  end
-->  xs=res(1:2,1:count-1);
-->  ys=res(3:4,1:count-1);
-->endfunction

-->w=rewrite(Irewrite0,'F',7);                  ← 3 niveaux de ré-écriture à partir de F
-->[xs,ys]=calcul_segments(w,60,1);
-->rect=[mini(xs),mini(ys),maxi(xs),maxi(ys)]*1.1
 rect  =

! - 3.461D-13    0.    121.93638    140.25 !

-->xsetech(frect=rect);
-->xsegs(xs,ys);                                   ← Voir Figue 3.1
```

3.0.3 Dessiner un arbre

Pour pouvoir générer des structures arborescentes on va enrichir notre alphabet $\mathcal{A}$ de deux symboles [et]. Leur signification est la suivante : quand on rencontre un symbole [il faut sauver la position ou l'on se trouve dans une pile (empiler). Quand on lit un symbole] il faut reprendre comme position courante la position qui était au sommet de la pile et rétirer cette valeur de la pile (dépiler). Par exemple, la chaîne l'interprétation de la chaîne `F[F]-F+` doit dessiner une structure en forme de Y.

On étends le programme de calcul des segments précédents pour qu'il interprète les symboles [et]. On sauvegarde aussi dans la pile, en même temps que le point courant, un compteur qui va nous servir à compter le nombre d'embranchement qui conduit à un sous arbre. On utilisera cela pour le graphique ultérieurement.

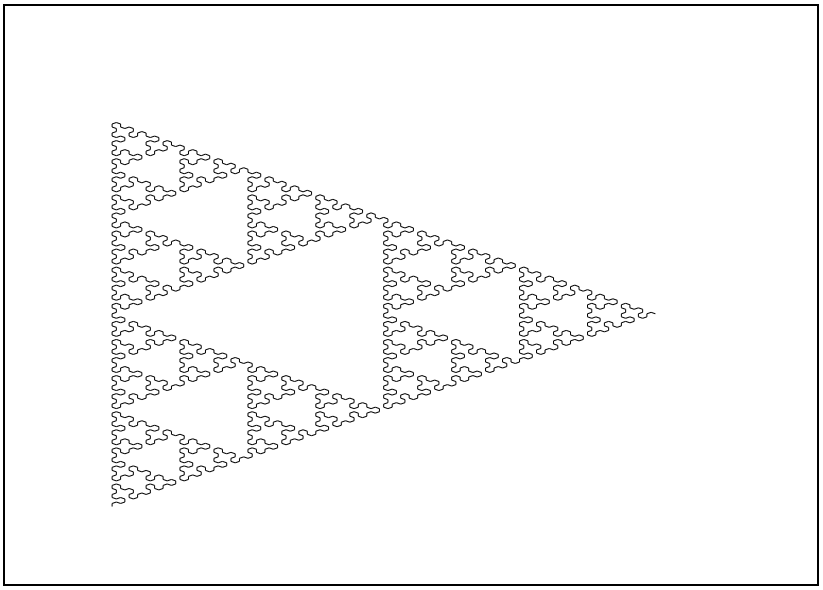

FIG. 3.1 – Premier modèle

```
-->function [xs,ys,ordre]=calcul_segments(w,delta,l)
-->  delta = delta*%pi/180;
-->  Rm = [cos(delta),-sin(delta);sin(delta),cos(delta)]
-->  Rp = [1,-1;-1,1].*Rm;
-->  L=length(w);
-->  stack=ones(5,L);                                  ← Matrice utilisée comme pile
-->  stc=1;                                  ← Premier indice utilisable dans la pile
-->  res=ones(5,L);                         ← Matrice pour stocker les segments calculés
-->  count=1;                       ← Premier indice utilisable pour stocker un segment
-->  pt=[0;0];dir=[0;1];
-->  ordre=1;                                        ← Compteur d'arborescence
-->  for i=1:length(w)
-->    select part(w,i)
-->     case '[' then
-->      stack(:,stc)=[pt;dir;ordre];stc=stc+1;     ← Il faut empiler les valeurs courantes
-->      ordre=ordre+1;
-->     case ']' then
-->      stc=stc-1;xx=stack(:,stc);                 ← Il faut dépiler les valeurs courantes
-->      pt=xx(1:2); dir=xx(3:4); ordre=xx(5);
-->     case '+' then
-->      dir= Rp*dir;                                                  ← Rotation +
-->     case '-' then
-->      dir= Rm*dir;                                                  ← Rotation -
-->    else
-->      newpt=pt + l*dir;                                ← Calcul d'un nouveau point
-->      res(:,count)=[pt(1);newpt(1);pt(2);newpt(2);ordre];
-->      pt = newpt;
-->      count=count+1;
-->    end
-->  end
```

```
-->  xs=res(1:2,1:count-1);
-->  ys=res(3:4,1:count-1);
-->  ordre=res(5,1:count-1);
-->endfunction

-->function [w]=Irewrite1(v)
-->  w=strsubst(v,'F','F[+F]F[-F]F');
-->endfunction

-->stacksize(10000000);
-->w=rewrite(Irewrite1,'F',5);
-->[xs,ys,ordre]=calcul_segments(w,30,1);
-->rect=[mini(xs),mini(ys),maxi(xs),maxi(ys)]*1.1
 rect  =

! - 56.747303    0.    44.55    267.3 !

-->xsetech(frect=rect);
-->xsegs(xs,ys);                                          ← Voir Figue 3.2
```

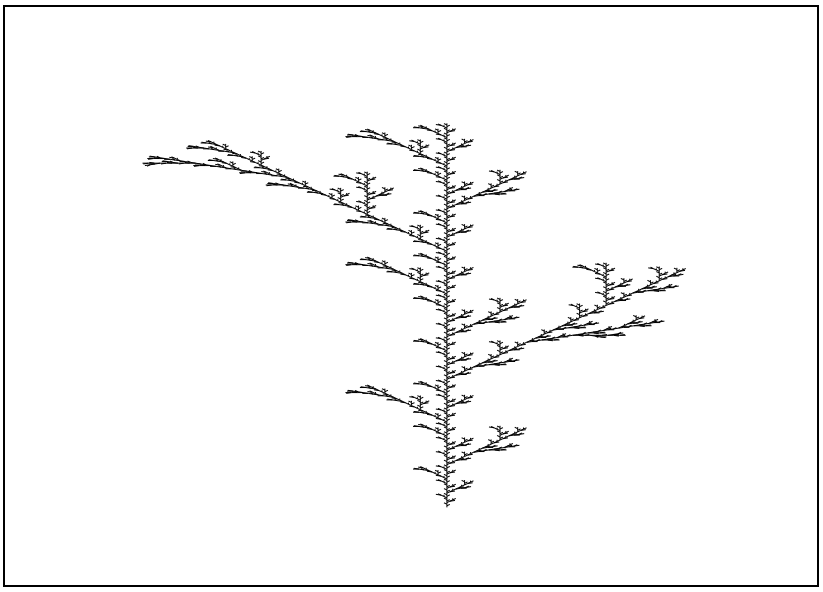

FIG. 3.2 – Un modèle d'arbre

On utilise maintenant pour chaque segment la valeur ordre pour changer l'épaisseur et la couleur du segment à tracer. On choisit les couleurs dans une tables de couleur dans les verts allant du noir au vert clair.

```
-->function []=dessiner(xs,ys,ordre)
-->  rect=[mini(xs),mini(ys),maxi(xs),maxi(ys)]*1.1
-->  xsetech(frect=rect);                              ← On fixe l'echelle courante
-->  m=maxi(ordre);
```

```
-->  if m <> 1 then
-->    cmap=[((1:m).^2)/m;1:m;((1:m).^2)/m]'/m;
-->    xset("colormap",cmap)                          ← On change de table de couleur
-->  end
-->  is=size(xs,'c');
-->  tm=maxi(ordre);
-->  for i=mini(ordre):maxi(ordre)
-->    I=find(ordre==i);                          ← On selectionne les segments d'ordre i
-->    xset("thickness",1*(tm-ordre(I(1))));      ← Épaisseur du trait fonction de l'ordre
-->    xsegs(xs(:,I),ys(:,I),i)                   ← Les segments d'ordre i avec la couleur i
-->  end
-->endfunction

-->function [w]=Irewrite2(v)
-->  w=strsubst(v,'H','F-[[H]+H]+F[+FH]-H');
-->  w=strsubst(w,'F','FF');
-->endfunction

-->set('old_style','on');                          ← On selectionne l'ancien mode graphique

-->w=rewrite(Irewrite2,'H',5);
-->[xs,ys,ordre]=calcul_segments(w,40,1);
-->dessiner(xs,ys,ordre);                                        ← Voir Figue 3.3
```

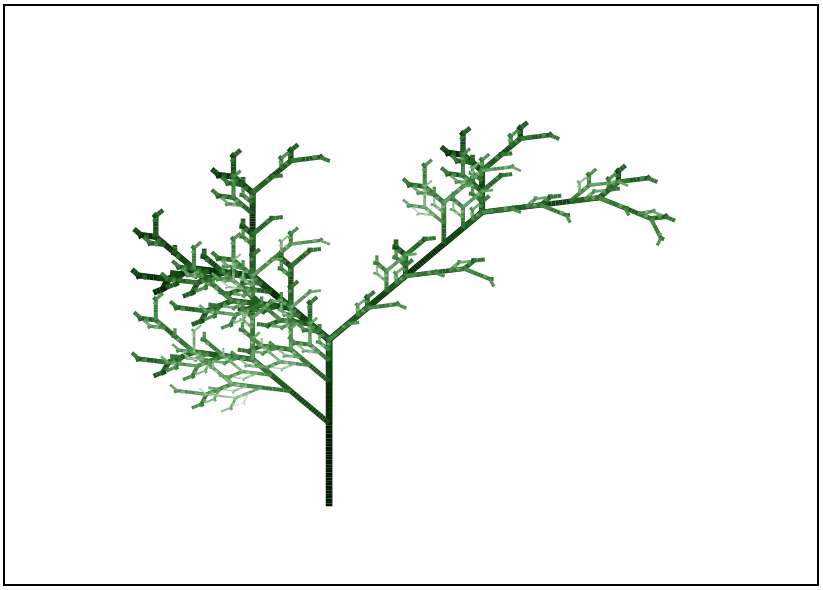

FIG. 3.3 – Un modèle d'arbre

Le défaut des DOL-systèmes vus précédemment est que pour un niveaux N de réécriture fixé les figures générées sont toujours identiques. On peut étendre ce qui précède en retirant l'aspect déterministe, on s'autorise alors à avoir plusieurs règles de réécriture pour une même. On se donne alors des probabilité de choix des régles que l'on utilise au moment de l'application des réécritures.

```
-->function [w]=Irewrite3(v)
-->  rew=['F[+F]','F[-F]','F[-F]F','F[+F]F']
-->  w="";
-->  for i=1:length(v)
-->    if part(v,i)== 'F' then
-->      w = w + rew(grand(1,'uin',1,4));
-->    else
-->      w = w +part(v,i)
-->    end
-->  end
-->endfunction

-->set('old_style','on');                    ← On selectionne l'ancien mode graphique

-->w=rewrite(Irewrite3,'F',9);
-->[xs,ys,ordre]=calcul_segments(w,32,1);
-->dessiner(xs,ys,ordre);                    ← Voir Figue 3.4
```

FIG. 3.4 – Un modèle d'arbre

Chapitre 4

Programmer : Musique serielle

Le but de ce T.P de programmation Scilab est de générer des partitions musicales en utilisant (librement) les règles de la musique sérielle ou en faisant générer la melodie par un système dynamique. On utilise Scilab pour générer un fichier `pmx`. Puis à partir de ce fichier des utilitaires (Unix ou Windows) permettent d'obtenir une partition musicale où un fichier Midi. Nous aurons donc besoin des utilitaires suivants :

- `pmx` qui permet de traduire un fichier `pmx` en format Midi ou en format MusicTEX .
- `musixtex` qui permet d'obtenir une partition (par exemple au format `pdf`) à partir d'un fichier MusicTEX.

Le rôle de Scilab est de construire une fichier MusicTEX.

4.1 Musique sérielle

On utilise des règles formulées par Arnold Schönberg pour construire notre partition musicale :

- Une phrase musicale (dite aussi serie de Schönberg) de départ P_1 est d'abord donnée, elle consiste en la donnée dans un ordre quelconque des douze sons de l'échelle chromatique tempérée, chacun de ces sons n'étant présenté qu'une fois. Nous utiliserons dans Scilab la fonction `grand` pour générer aléatoirement cette série.
- A partir de cette première phrase musicale on en génère trois autres.
  - Une phrase récurrente P_2, obtenue en renversant l'odre des notes de la phrase P_1 de la dernière à la première note.
  - Une phrase renversée P_3 obtenue en changeant la direction des intervalles de la phrase P_1
  - Une phrase récurrente renversée P_4 obtenue en changeant la direction des intervalles de la phrase P_2

On peut ensuite transposer sur les onze autres degrés de l'échelle chromatique les quatres phrases initiale $(P_i)_{i=1,\dots,4}$. On obtient donc au final 48 formes que l'on peut utiliser pour composer un morceau, ce que nous ferons de façon rudimentaire en tirant aléatoirement des phrases succéssives dans les 48 possibles. On peut encore jouer sur les rythmes et les auteurs de notes (équivalence de toutes les notes du même nom entre elles).

4.1.1 Générer les 48 phrases possibles

Comme il y a $N = 12$ notes dans la gamme chromatiques nous allons pour l'instant dans la phase de génération les coder par des entiers $1, \ldots, N$. Nous nous limitons donc à des notes dans un même octave. Les transformation proposées peuvent faire sortir de l'intervalle $1, \ldots, N$ et on s'y ramenera en utilisant une transposition d'octave apropriée.

Voila les fonctions de base pour générer les phrases initiales

```
-->function germe=initiale()                    ← Tirage aléatoire d'une série de Schönberg
-->  N=12;
-->  notes=(1:N)';
-->  germe=grand(1,'prm',notes) ;          ← On tire une permutation aléatoire de 1,...,N
-->endfunction

-->function y=renverse_ecarts(x)
-->  // renversement des ecarts
-->  ecart=diff(x);                  ← Calcul des écarts succéssifs en nombre de demi-tons
-->  y = cumsum([x(1);-ecart]);                          ← La série des écarts renversés
-->endfunction

-->function y=recurrente(x)
-->  y=x($:-1:1)                                        ← La série renversée en temps
-->endfunction

-->function y=moduloN(y)                  ← On ramène les notes dans l'intervalle 1,...,N
-->  N=12;
-->  y = pmodulo(y,N);
-->  y(y==0) = N
-->endfunction
```

Que l'on utilise maintenant pour générer les 48 phrases utilisables sous forme d'une matrice $N \times 48$:

```
-->germe=initiale();              ← Phrase initiale codée sous la forme d'un vecteur colonne

-->pos=[germe,renverse_ecarts(germe),...                    ← Les quatres phrases as-
sociées codées sous forme d'une matrice
-->     recurrente(germe),renverse_ecarts(recurrente(germe))];

-->pos=moduloN(pos);                                  ← On ramène les notes dans 1,...,N

-->N=12;
-->all=[pos];
-->for i=1:N-1
-->  all=[all,moduloN(pos+i)];          ← Toutes les transpositions de nos quatres phrases
-->end

-->size(all)                                          ← Une phrase par colonne et 48 colonnes
 ans  =
```

```
!   12.    48. !
```

4.1.2 Le codage des notes en pmx

Les 12 notes de la gamme chromatique (à octave fixé) sont codées en `pmx` par des chaînes de caratères que nous codons ici dans un vecteur ligne de chaînes de caratères Scilab :

```
-->notes_m = ['cn','cs','dn','ds','e','fn','fs','gn','gs','an','as','b'] ;
-->notes_m = notes_m + '44';                ← On fixe la durée et l'octave de façon absolue
```

Nous allons générer ici un petit morceau de 8 mesure et chaque mesure va contenir une phrase possible (sans répétition) parmi les 48 possibles obtenue à nouveau par tirage aléatoire de loi uniforme sur les 48 phrases. Comme chaque phrase contient 12 notes, nous allons générer 3 triolets de croches par mesures. Un trilet de croches se code en `pmx` sous la forme d'une chaîne de caractère `Ax3 B C`, où `A`, `B` et `C` sont les codes des trois notes qui composent le triolet. Une mesure se termine par le caratère `/`.

Nous écrivons une fonction qui renvoie un vecteur colonne de chaîne de caratères. Chaque ligne du vecteur codant une mesure du morceau.

```
-->function [txt]=morceau(all)
-->  p=size(all,'c');                                                    ← p vaut 48 !
-->  I=grand(1,'prm',(1:p)') ;                  ← Une permutation aléatoire des 48 phrases
-->  pat=I(1:8)';                                                  ← On n'en garde que 8
-->  txt=string([]);
-->  for i=pat
-->    im = notes_m(all(:,i)) ;          ← On transforme la phrase i en vecteur de chaîne
-->    im(1:3:$) = im(1:3:$) +'x3';              ← Toutes les 3 notes, on forme un triolet
-->    im= strcat([im,'/'],' ');              ← On concatène les éléments de im et on ra-
joute le code de fin de mesure
-->    txt=[txt;im];                  ← On rajoute le code de la nouvelle mesure dans txt
-->  end
-->endfunction
```

Pour obtenir le codage du morceau en `pmx` il ne reste plus qu'à rajouter en entête fixe et à écrire le tout dans un fichier.

```
-->txt=morceau(all);                          ← Un morceau de 8 mesures choisies dans all

-->head=['1 1 4 4 4 40.000';' 0 0 2 16 0.0';' '
-->      't'
-->      './'
-->      'Aa1br'
-->      'It60';'w140m'];

-->mputl([head;txt],'grand.pmx')                        ← Entête et écriture dans un fichier
-->unix('pmx grand')                                    ← Il ne reste plus qu'à utiliser pmx
```

```
ans  =

    0.
```

La partition générée est alors la suivante :

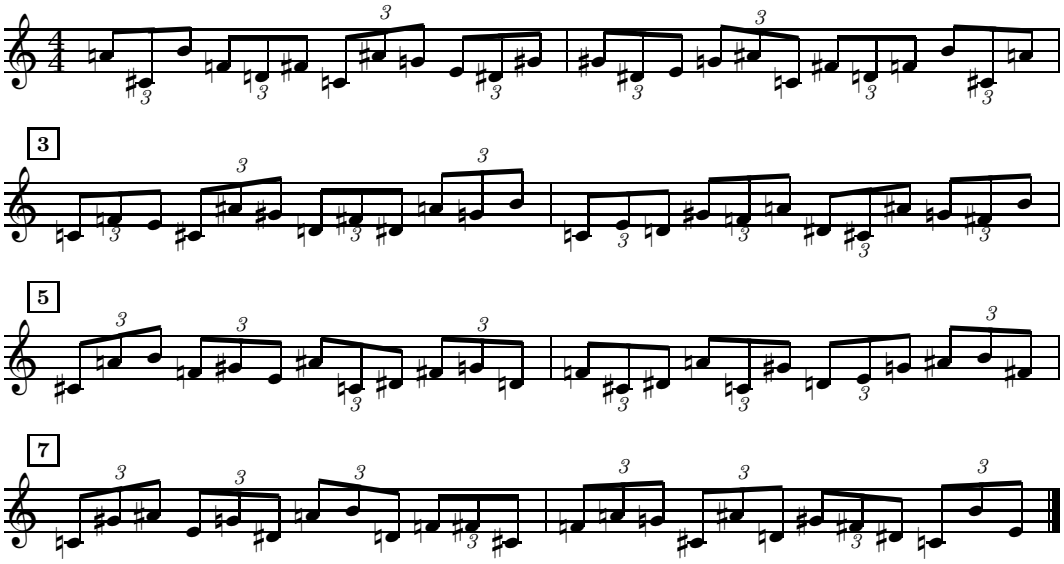

FIG. 4.1 – Partition générée

4.2 Utilisation du système de Lorenz

```
-->//compute lorenz differential eqn solution
-->y=ode(1.e-8*[1;1;1],0,0:1:100,'loren');
-->[n1,n2]=size(y) ;

-->notes=y(1,:);
-->mi=mini(notes);mx=maxi(notes);

-->notes_b = ['c','d','e','f','g','a','b'] ;
-->notes_n = [notes_b + '44', notes_b + '45'];
-->notes_c = [notes_b + '84', notes_b + '85'];

-->N=size(notes_n,'*');

-->v=y(2,:);
-->vi=mini(abs(v));vx=maxi(abs(v));
-->I=find( abs(v) >= vi+ (vx-vi)/2);

-->notes= int(1+ (notes-mi)*(N-1)/(mx-mi));

-->all_n = notes_n(notes);
```

```
-->all_c = notes_c(notes);
-->all = all_n;
-->all(I)= all_c(I);
-->d = ones(all_n);
-->d(I)= 0.5;

-->// On regroupe les notes par mesure

-->count=1;
-->mes=string([]);
-->while %t
-->  t=cumsum(d(count:$));
-->  i=find(t > 4);
-->  if i<>[] then
-->    ti= sum(d(count:count+i(1)-2));
-->    im = all(count:count+i(1)-2);
-->    if ti == 3.5 then
-->      im= strcat([im,'r8','/'],' ');
-->    else
-->      im= strcat([im,'/'],' ');
-->    end
-->    mes=[mes;im];
-->    count=count+i(1)-1;
-->  else
-->    break
-->  end
-->end

-->head=['1 1 4 4 4 40.000 0';' 0 4 16 0.0';' '
-->      't'
-->      './'
-->      'Aa1br'
-->      'It260';'w140m'];

-->mputl([head;mes],'lorenz.pmx')

-->unix('pmx lorenz.pmx');
-->unix('timidity -Ow  lorenz.wav lorenz.mid');
```

La partition générée est alors la suivante :

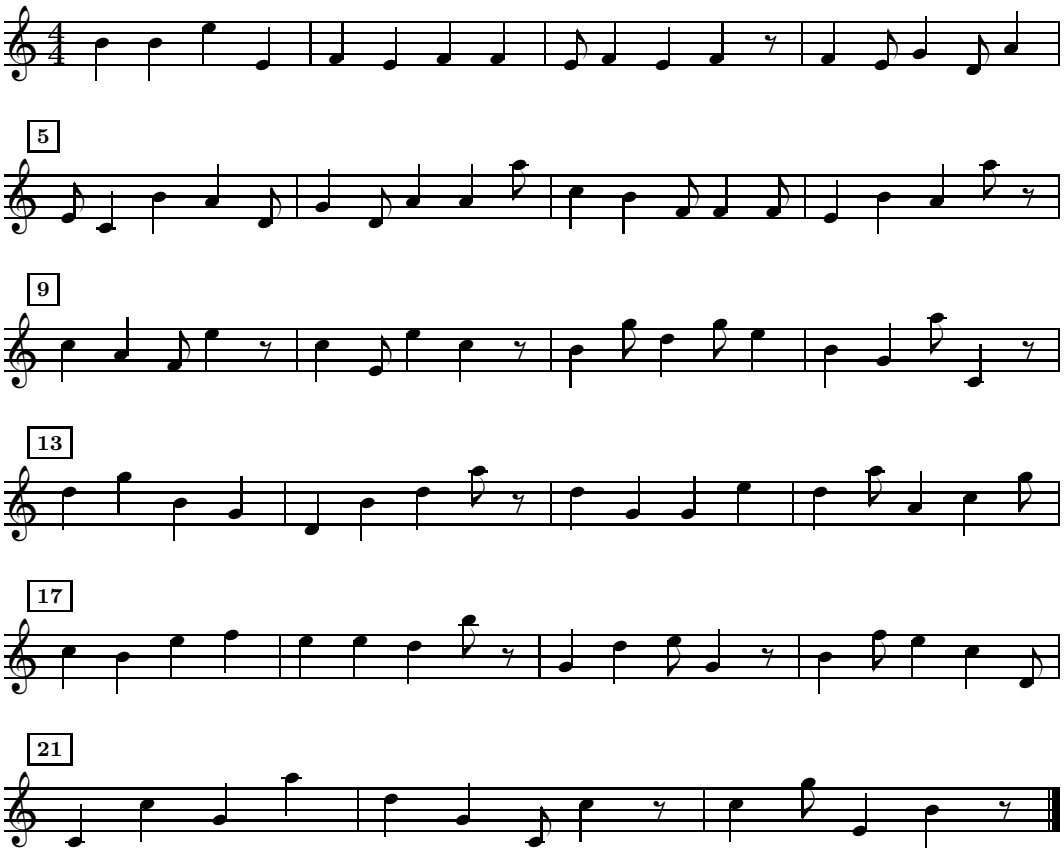

FIG. 4.2 – Partition générée

Bibliographie

[1] N.Janey. *Modélisation et synthèse d'images d'arbres et de bassins fluviaux associant méthodes combinatoires et plongement automatique d'arbres et cartes planaires.* PhD thesis, Thèse de l'Université de Franche-Comté, `http://raphaello.univ-fcomte.fr/These/Default.htm`, 1992.

[2] P.Prusinkiewicz and A.Lindenmayer. *The Algorithmic beauty of plants.* Springer, 1996.

[3] X.G.Viennot, G.Eyrolles, N.Janey, and D.Arquès. Combinatorial analysis of ramified patterns and computer imagery of trees. *Computer Graphics*, 23 :31–40, 1989.

Index