Une introduction à la programmation par Scilab

Seconde Partie

Aspects Avancés

Jean-Jacques Girardot

École des Mines de Saint-Etienne
158 Cours Fauriel
42023 Saint-Etienne.

Mots-clefs : introduction à la programmation, Scilab
Seconde Partie. Date de création : 4 octobre 2000.

Table des matières

I Aspects Avancés 9

1 Graphiques 11
- 1.1 Tour d'horizon . . . 11
- 1.2 Une petite introduction au graphisme . . . 13
  - 1.2.1 Le matériel . . . 13
  - 1.2.2 Les concepts . . . 14
- 1.3 Les bases du graphisme en Scilab . . . 14
  - 1.3.1 Fenêtres . . . 14
  - 1.3.2 Exercice . . . 15
  - 1.3.3 Fenêtres multiples . . . 15
  - 1.3.4 Contexte graphique . . . 15
    - 1.3.4.1 Géométrie . . . 15
- 1.4 (Re) Tours de Hanoi . . . 16
  - 1.4.1 Initialisation du système graphique . . . 16
  - 1.4.2 Tracés et animation . . . 17
  - 1.4.3 Mise en place des éléments . . . 18
- 1.5 Graphique 3D . . . 19
- 1.6 Exercices . . . 20
  - 1.6.1 Démonstration . . . 20
  - 1.6.2 Tableau de fils . . . 20
  - 1.6.3 Palettes graphiques . . . 21
  - 1.6.4 Triangle de Serpinsky . . . 21
  - 1.6.5 Flocon de Koch . . . 22
  - 1.6.6 Courbe du dragon . . . 22
  - 1.6.7 La tortue Logo . . . 24

2 Les chaînes de caractères 25
- 2.1 Présentation . . . 25
- 2.2 Qu'est-ce qu'un caractère ? . . . 27
  - 2.2.1 Les caractères en Scilab . . . 27
  - 2.2.2 Autres représentations des caractères . . . 28
- 2.3 Manipulation des chaînes de caractères . . . 28
  - 2.3.1 Concaténation . . . 28
  - 2.3.2 Extraction de sous-chaîne . . . 30
  - 2.3.3 Recherche de sous-chaîne . . . 30

2.3.4 Remplacement de chaînes . . . 31
2.3.5 Élimination de blancs . . . 31
2.3.6 Transformations majuscules/minuscules . . . 32
2.4 Divers . . . 32
2.4.1 Lecture de caractères . . . 32
2.4.2 Chaînes et paramètres . . . 33
2.5 Tableaux . . . 33
2.6 Applications . . . 34
2.7 Exercices . . . 35
2.7.1 Chiffres Romains . . . 35
2.7.2 Palindromes . . . 36
2.7.3 Dictionnaire . . . 36
2.7.4 Anagramme . . . 37
2.7.5 Conversions . . . 37

3 Listes et matrices creuses 39
3.1 Les listes : présentation . . . 39
3.1.1 Description des listes . . . 39
3.1.2 Création de listes . . . 39
3.2 Opérations sur listes . . . 40
3.2.1 Taille . . . 40
3.2.2 Indexation . . . 40
3.2.3 Addition et destruction d'éléments . . . 40
3.2.4 Extraction . . . 41
3.2.5 Concaténation de listes . . . 42
3.2.6 Note sur l'efficacité des diverses opérations . . . 42
3.3 Matrices creuses . . . 43
3.3.1 Présentationsont . . . 43
3.3.2 Quelques opérations . . . 44
3.4 Exercices . . . 44
3.4.1 Tableaux creux . . . 44

4 À l'intérieur de Scilab 47
4.1 Types de données . . . 47
4.1.1 Présentation . . . 47
4.1.2 Quelques types de données . . . 47
4.1.2.1 Tableaux numériques . . . 47
4.1.2.2 Tableau de polynômes . . . 48
4.1.2.3 Tableaux booléens . . . 49
4.1.2.4 Tableaux creux . . . 50
4.1.2.5 Tableaux d'entiers . . . 50
4.1.2.6 Tableaux de chaînes de caractères . . . 50
4.1.2.7 Fonctions au format source . . . 50
4.1.2.8 Fonction compilée . . . 50
4.1.2.9 Bibliothèque de fonctions . . . 50

4.1.2.10 Tableaux généralisés, dits «listes» . . . 50
4.1.2.11 Tableaux généralisés typés, dits «listes typées» . . . 50
4.1.2.12 Pointeurs . . . 50
4.2 Nombres aléatoires . . . 50
4.3 Utilisation de programmes extérieurs . . . 50
4.4 Exercices . . . 52

II Annexes 53

A Scilab, le bétisier 55
A.1 Les plaisirs de la notation . . . 55
A.1.1 Syntaxe . . . 55
A.1.2 Transposition . . . 56
A.1.3 Les expressions mystérieuses . . . 56
A.1.4 Vecteurs . . . 56
A.1.5 De l'équivalence des chaînes . . . 57
A.2 Les joies des variables . . . 57
A.2.1 Variables de contrôle de boucle . . . 57
A.2.2 Une variable est-elle définie ou non ? . . . 57
A.2.3 De l'effacement des variables . . . 58
A.3 Des opérations sur tableaux . . . 58
A.3.1 Le tableau vide . . . 58
A.3.2 Création de vecteurs . . . 59
A.4 De la nature des fonctions . . . 61
A.4.1 Leur création . . . 61
A.4.2 De l'inégalité des fonctions . . . 61
A.4.3 De l'affectation des fonctions . . . 62

B Principales commandes de Scilab 65
B.1 Information . . . 65
B.2 Gestion de l'espace de travail . . . 65
B.3 Contrôle divers . . . 66
B.4 Mise au point . . . 66
B.5 Variables du système . . . 66

C Solution de quelques exercices 67
C.1 Exercices du chapitre 1 . . . 67
C.1.1 Hypoténuse . . . 67
C.1.2 Disque . . . 67
C.1.3 Nombres complexes . . . 67
C.1.4 Affichage . . . 69
C.1.5 Approximations . . . 70
C.1.6 Tracé de courbes . . . 70
C.1.7 Interface . . . 70

C.1.8 Programme nu . . . 71
C.1.9 Menus . . . 71
C.1.10 Compatibilité . . . 71
C.2 Exercices du chapitre 2 . . . 72
C.2.1 Éléments de syntaxe . . . 72
C.2.2 Quelques calculs . . . 73
C.2.3 Nombres premiers . . . 74
C.2.4 Nombres parfaits . . . 75
C.2.5 PGCD . . . 76
C.3 Exercices du chapitre 3 . . . 77
C.3.1 Complexités et puissances des machines . . . 77
C.3.2 Évaluation de la complexité d'algorithmes . . . 78
C.3.3 Un peu de calcul . . . 79
C.3.4 Sous-séquence maximale . . . 79
C.4 Exercices du chapitre 4 . . . 81
C.4.1 Lecture . . . 82
C.4.2 Notation . . . 82
C.4.3 Construction de matrice . . . 82
C.4.4 Oeil gauche . . . 83
C.4.5 Matrices de Hilbert . . . 83
C.4.6 Petit utilitaire . . . 84
C.4.7 Produit scalaire . . . 84
C.4.8 Tri ascendant . . . 85
C.4.9 Réversion d'un vecteur . . . 85
C.5 Exercices du chapitre 5 . . . 85
C.5.1 Triangle de Pascal . . . 85
C.5.2 Notations . . . 86
C.5.3 Nombres premiers . . . 87
C.5.4 Intégration de fonctions . . . 88
C.6 Exercices du chapitre 6 . . . 90
C.6.1 Gags . . . 90
C.6.2 L'indice erroné . . . 91
C.6.3 Meilleure expression . . . 91
C.6.4 Diagonale . . . 92
C.6.5 Tours de Hanoi . . . 92
C.6.5.1 Première partie . . . 92
C.6.5.2 Seconde partie . . . 94
C.7 Exercices du chapitre 7 . . . 96
C.7.1 Tableau de fils . . . 96
C.7.2 Palettes graphiques . . . 96
C.7.3 Triangle de Serpinsky . . . 97
C.7.4 Courbe du Dragon . . . 98
C.7.5 La tortue Logo . . . 99
C.8 Exercices du chapitre 8 . . . 99

C.8.1 Chiffres romains . . . 99
C.8.2 Palindromes . . . 101
C.9 Exercices du chapitre 9 . . . 102
C.9.1 Tableaux creux . . . 102
C.10 Exercices du chapitre 10 . . . 103
C.11 Tours de Hanoi . . . 103

Table des figures **113**

Références **115**

Index **117**

Première partie

Aspects Avancés

Chapitre 1

Graphiques

Nous présentons dans ce chapitre certains aspects des outils graphiques disponibles dans SCILAB. Cette introduction (comme le reste de ce cours, d'ailleurs), ne vise pas à être exhaustive, mais plutôt à expliquer les quelques notions de base indispensables pour faire bon usage de la documentation disponible.

1.1 Tour d'horizon

SCILAB propose un grand nombre de fonctions graphiques directement utilisables. Ainsi, la session suivante permet-elle d'ouvrir une fenêtre graphique, et d'y tracer une courbe, la fonction $y = x^2$:

```
-->plot((1 :0.1 :20)^2)
```

Il est ici fait appel à une fonction unique, `plot`. Le paramètre est un vecteur de 201 valeurs. La fonction `plot` ouvre une *fenêtre graphique* (c.f. figure 1.1, page 12), et y trace la fonction, en considérant que cette fonction est définie point par point, que les y sont définis par le vecteur fourni en paramètre, et que les x correspondants sont les valeurs du vecteur `1 :N`, `N` étant le nombre d'éléments du paramètre. La fonction `plot` trace entre les axes et y affiche une échelle (plus ou moins farfelue en l'occurrence, puisque l'axe des x affiche les *numéros* des éléments...). Elle propose d'autres modes opératoires, qui offrent un meilleur contrôle sur ce qui s'affiche. Par exemple,

```
-->V=(0 :0.1 :20) ;
-->plot(V,V^2)
```

permet d'afficher correctement les échelles en x et y ; il est même possible de placer des étiquettes sur les axes ainsi qu'un titre au graphique avec une écriture telle que :

```
plot(V, V^2, 'x', 'y', 'Fonction y=x^2')
```

Le même résultat peut s'obtenir par la succession des deux appels :

```
-->plot(V, V^2)
-->xtitle('Fonction y=x^2', 'x', 'y')
```

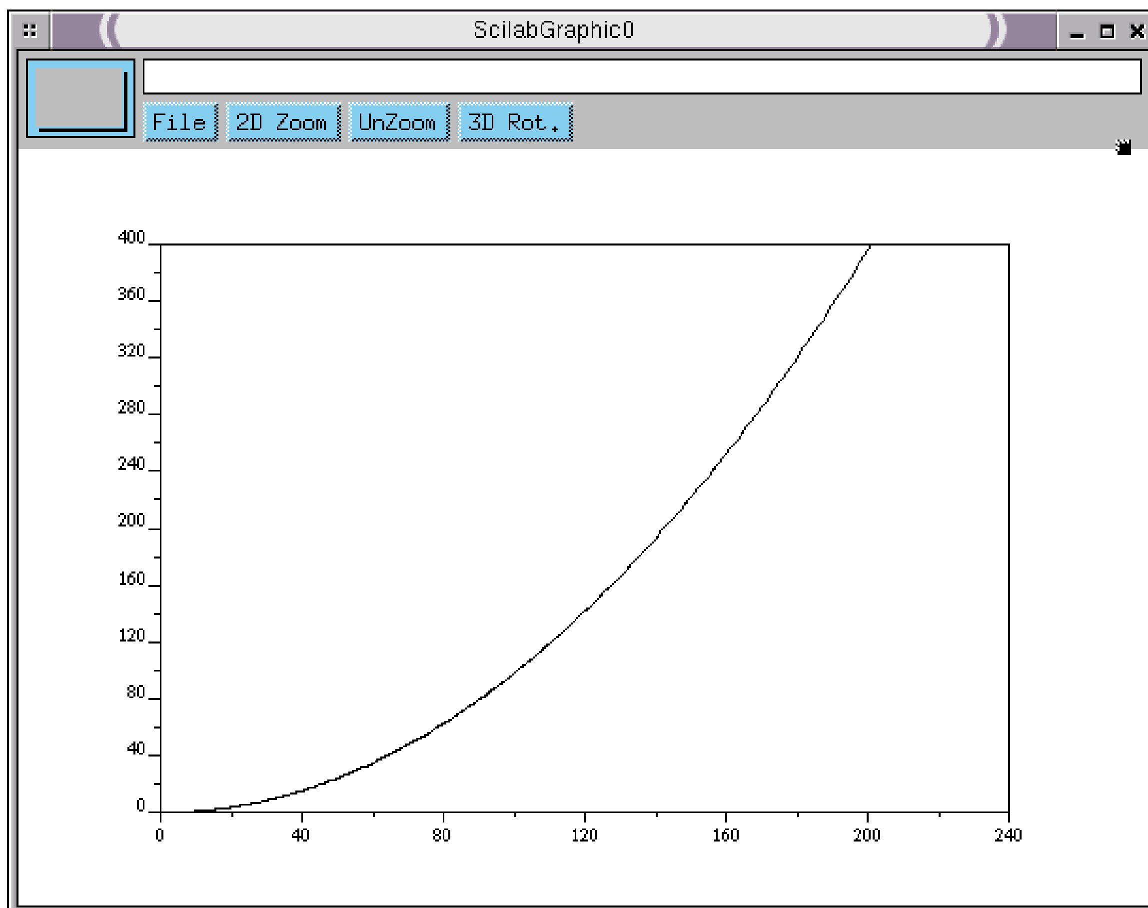

FIG. 1.1 – Fonction $y = x^2$

De manière générale, la plupart des besoins courants sont satisfaits avec un petit nombre d'opérations graphiques prédéfinies, dont :

```
help Graphics
```

nous fournit la liste.

1.2 Une petite introduction au graphisme

L'objectif de cette partie n'est pas de faire le tour de concepts fondamentaux du graphisme. Le lecteur trouvera, dans des ouvrages comme [1], de bonnes introductions au domaine. Nous nous contenterons ici de donner quelques indications générales sur les aspects matériels et logiciels du graphisme.

1.2.1 Le matériel

Les *écrans graphiques* conçus aujourd'hui sont des matrices rectangulaires, dont chaque point, dit *pixel* (pour «picture element») peut afficher une couleur déterminée par une valeur numérique associée au pixel. En général, il y a trois composantes, pour chacune des trois couleurs de base de l'écran (typiquement, rouge, bleu et vert, le jaune étant obtenu par addition de vert et de rouge), l'intensité de chaque couleur étant représentée par un nombre, le plus souvent compris entre 0 (intensité nulle) et 255 (intensité maximale). Les intensités des couleurs peuvent donc être représentée sur 8 bits, et l'information associée à chaque pixel occupe donc 24 bits. Chaque pixel peut ainsi afficher une couleur parmi 2^{24}, soit plus de 16 millions de couleurs différentes.

Les tailles des écrans varient entre 300×200 pixels, pour de petits écrans, jusqu'à 1600×1200 pixels pour des écrans haut de gamme, avec des tailles intermédiaires telles que 600×400, 1024×780, et 1280×1024.

La *résolution* d'un écran va également dépendre de la taille des pixels. Ceux-ci sont assimilés à des carrés, la résolution s'exprimant souvent en nombre de pixels par pouce, qui est typiquement de l'ordre de 70 pour les écrans d'ordinateurs (cette résolution, pour une imprimante laser, est de l'ordre de 300 à 600 points par pouce).

L'écran est connecté à une carte graphique, qui va fournir des opérations d'un peu plus haut niveau que le simple positionnement de la couleur d'un pixel. On trouvera :

- le tracer de segments (avec calcul des couleurs des pixel, utilisation de méthode d'anti-alliassage pour éviter les phénomènes d'escalier) ;
- le remplissage de polygones (couleur unie, éventuellement dégradés) ;
- des opérations pour associer une texture à un polygone en 3D, et projeter celui-ci sur le plan de l'écran ;
- la gestions des faces cachées, dans le cas du 3D.

Les cartes graphiques peuvent être extrêmement complexes ; la société Silicon Graphics, par exemple, vend des ordinateurs dont le processeur graphique est beaucoup plus coûteux que l'unité centrale.

Signalons aussi qu'il commence à exister des normes décrivant les opérations d'une carte graphique : OPEN-GL décrit un ensemble de fonction pour la représentation d'objets en 3D.

Enfin, des cartes graphiques spécialisées pour la restitution de la vidéo existent ; elle implantent la norme MPEG.

1.2.2 Les concepts

Il est clair que la réalisation de graphismes, au moins lorsqu'elle se fait à un niveau assez bas, c'est-à-dire assez proche du matériel, nécessite de bonnes connaissances en algorithmique et en géométrie : comment calculer l'intersection de deux droites, de deux plans, d'une droite et d'un plan, comment détecter si un point est visible ou non, comment tracer la partie de la droite d'équation $ax + by + c = 0$ située dans un rectangle donné, etc. Il existe des algorithmes très spécialisés, qui permettent de rendre relativement performantes ces opérations qui doivent être réalisées en très grand nombre pour la synthèse d'une image.

1.3 Les bases du graphisme en Scilab

Là encore, nous ne désirons donner qu'un bref aperçu des éléments de base du graphisme en Scilab. Le lecteur trouvera naturellement une description détaillée de l'ensemble des opérations graphiques du langage dans la documentation en ligne.

1.3.1 Fenêtres

Scilab effectue ses opérations graphiques à l'intérieur de fenêtres spéciales, les *fenêtres graphiques*, différentes de la *fenêtre d'interaction*. Ces fenêtres conservent une mémoire des opérations qui y ont eu lieu ; des modifications éventuelles de la géométrie de ces fenêtres (largeur, hauteur) se traduiront par un nouveau tracé des éléments graphiques qu'elles contiennent.

Par défaut, la fenêtre graphique permet d'afficher la partie du plan comprise entre 0 et 1 en x et en y. Le programme suivant permet d'afficher le carré encadrant cette portion :

```
function gf01()
xbasc() ;
m1= [0 0 1 1; 0 1 1 0 ];
m2= [0 1 1 0; 1 1 0 0 ];
xsegs(m1,m2) ;
```

Le premier appel, `xbasc()`, réinitialise et affiche la fenêtre graphique. Les deux variables `m1` et `m2` vont contenir les coordonnées de quatre segments que va tracer la fonction `xsegs()`. Le premier paramètre de celle-ci est une matrice, qui va contenir autant de colonnes que de segments à tracer, et représente les abscisses des extrémités de ces segments : une valeur de la première ligne est l'abscisse de l'une des extrémités du segment, la valeur correspondante de la seconde ligne est l'abscisse de l'autre extrémité du même segment. Le second paramètre de la fonction joue un rôle identique en ce qui concerne les ordonnées.

On aura noté que les affichages s'effectuent, par défaut dans une fenêtre rectangulaire. Les axes sont donc orthogonaux, mais non orthonormés. Il est possible, au moyen de la fonction `square()`, de modifier la géométrie de l'affichage pour obtenir une fenêtre isométrique, en précisant éventuellement les limites en x et y de l'affichage :

```
square(0,0,1,1)
```

restructure la fenêtre graphique pour qu'elle soit carrée, et que la zone d'affichage recouvre la partie du plan dont les points ont des coordonnées comprises entre 0 et 1, en x comme en y.

Opération	Description
`xset("window",n)`	Sélection de la fenêtre de numéro n comme fenêtre courante
`xselect()`	Fait passer la fenêtre courante au premier plan
`xbasc(n)`	Efface la fenêtre n. Sans paramètre, efface la fenêtre courante
`xdel(n)`	Détruit la fenêtre n. Sans paramètre, détruit la fenêtre courante

Fig. 1.2 – Gestion des fenêtres graphiques

Opération	Description
`isoview`	Sélection d'axes orthonormés, ne modifiant pas la taille de la fenêtre
`square`	Sélection d'axes orthonormés, avec modification de la taille de la fenêtre
`xsetech`	Sélection d'une sous-fenêtre

Fig. 1.3 – Géométrie des fenêtres graphiques

1.3.2 Exercice

Complétez la fonction `gf01()` en tracant, toujours grâce à `xsegs()`, le cercle inscrit à l'intérieur du carré obtenu.

1.3.3 Fenêtres multiples

Scilab permet de gérer plusieurs fenêtres, bien qu'à un instant donné, il existe une fenêtre courante dans laquelle s'affichent les résultat des primitives utilisées. Le tableau 1.2, page 15, décrit les primitives gérant ces fenêtres.

1.3.4 Contexte graphique

À chaque fenêtre est associé un contexte graphique, c'est-à-dire un ensemble de paramètres qui déterminent les caractéristiques géométriques de la fenêtre, la palette de couleurs utilisée, les couleurs du fond, des traits, etc. La fonction `xset()` permet d'agir sur certaines de ces valeurs.

La syntaxe générale de cette fonction est *xset(nom,[valeurs])*, où *nom* est une chaîne de caractères qui décrit la caractéristique modifiée, les autres paramètres de la fonction représentant la ou les nouvelles valeurs à mettre en place. On consultera naturellement la documentation de cette fonction.

1.3.4.1 Géométrie

Quelques commandes, décrites dans le tableau 1.3, page 15, permettent d'agir sur la géométrie de la fenêtre. On notera que `xsetech` permet de sélectionner une partie de la fenêtre courante. Il devient possible d'afficher plusieurs dessins différents dans la même fenêtre, en créant chacun d'eux dans une telle sous-fenêtre.

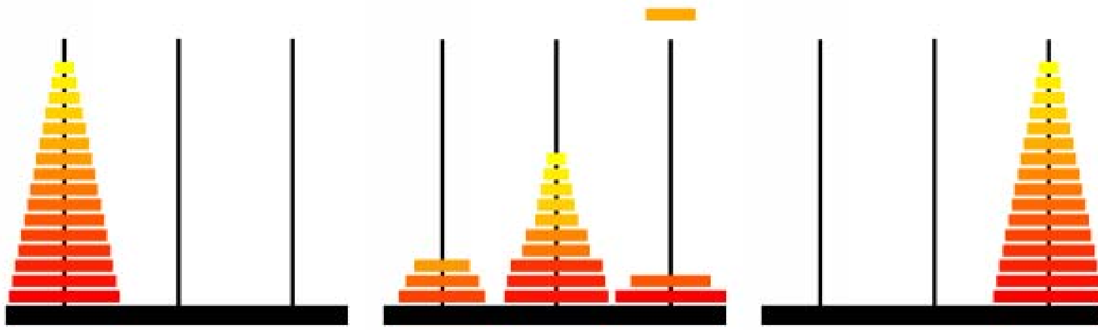

FIG. 1.4 – Tours de Hanoi

1.4 (Re) Tours de Hanoi

On se propose *d'animer* les tours de Hanoi, dont nous avions traité l'exemple au § ??. La figure 1.4, page 16, donne une idée de ce que nous désirons obtenir. La première figure montre la configuration initiale, ici 16 disques posés sur le piquet n°1. La seconde figure montre l'une des étapes intermédiaires (avec 16 disques, il n'y en a pas moins de $2^{16} - 1$!), l'un des disques passant du piquet n°1 au piquet n°3, à moins que ce ne soit l'inverse. La dernière figure montre la position finale, les 16 disques étant parvenus sur le dernier piquet.

Cette figure, ainsi que l'animation des déplacements des disques d'un piquet à un autre, est obtenue en utilisant la seule opération d'affichage de rectangles pleins, `xfrect()`.

Nous allons analyser ici certains des sous-problèmes auxquels il convient de faire face pour cette réalisation.

1.4.1 Initialisation du système graphique

La première opération consiste à choisir un *driver graphique*, c'est à dire un logiciel capable de recevoir des commandes d'affichage et de les traduire en *pixels*[1] sur une unité graphique (écran ou imprimante), selon une norme spécifique. SCILAB propose plusieurs de ces drivers, pour le système de fenêtrage X11 (les drivers «X11» et «Rec»), ou encore permettant de construire une représentation de l'image selon une norme telle que Postscript, Xfig ou Gif. Le driver «Rec» est similaire à «X11», avec la particularité d'enregistrer les commandes, ce qui permet de sauvegarder ensuite le dessin dans un fichier. En contrepartie, il s'avère devenir très lent lorsque de nombreux ordres lui sont donnés. Nous utiliserons donc le driver «X11», le choix étant fait par la commande `driver()` :

```
driver("X11") ;
```

L'initialisation des paramètres et l'affichage de la fenêtre graphique s'effectue par `xbasc()`, déjà présentée :

```
xbasc() ;
```

[1] Pixel : «picture element», c'est à dire un *point* d'un dessin, caractérisé entre autres par sa couleur.

Nous allons ensuite choisir une *palette graphique*, c'est-à-dire un ensemble de couleurs auxquelles nous pourrons faire référence dans notre dessin. Pour SCILAB, une palette graphique est un tableau de taille $n \times 3$, dont chaque ligne détermine une couleur par les valeurs, comprises entre 0 et 1, des trois composantes primaires qui sont le *rouge*, le *vert* et le *bleu*. Le système fonctionne en mode additif, l'addition de *rouge* et de *vert* donnant du *jaune*, l'addition des trois couleurs donnant le *blanc*, etc. Par exemple, l'instruction suivante définit une palette contenant dans l'ordre les couleurs *noir*, *bleu*, *vert*, *rouge*, *orange*, *jaune* et *blanc* :

```
palette=[0 0 0 ; 0 0 1 ; 0 1 0 ; 1 0 0 ; 1 0.5 0 ; 1 1 0 ; 1 1 1]
```

Il faut naturellement indiquer à SCILAB que c'est cette palette de couleur qui va être utilisée pour les affichages ; on utilise pour ce faire la commande :

```
xset("colormap",palette)
```

Les couleurs seront ultérieurement définie par leur numéro de ligne dans cette matrice, 1 pour *noir*, 4 pour *rouge*, etc... La sélection d'une couleur s'effectue par :

```
xset("pattern",4)
```

pour choisir, par exemple, la quatrième couleur de notre palette.

On indique enfin que l'on va opérer dans un repère orthonormé, et l'on précise les coordonnées du point en bas à gauche et en haut à droite, par exemple :

```
square(0,0,1000,1000) ;
```

1.4.2 Tracés et animation

Nous utiliserons, pour tous les tracés, la primitive `xfrect(x,y,l,h)` qui permet d'afficher un rectangle plein défini par les coordonnées du coin haut gauche, x et y, la largeur l et la hauteur h. La couleur de remplissage est elle-même déterminée par un appel `xset("pattern",numéro)`.

Comment animer un objet ? Le problème est aisé si l'on se contente de déplacer des rectangles horizontalement ou verticalement. Les rectangles étant de couleur uniforme, il suffit d'effacer une petite partie à droite, de rajouter à gauche une partie de même taille, pour donner l'impression que le rectangle s'est déplacé vers la gauche. Pour effacer, il suffit de tracer un rectangle de la même couleur que le fond. Le programme ci-dessous initialise la fenêtre graphique, y trace un cadre, et y fait se déplacer un rectangle :

```
function gf06()
driver("X11") ;
xbasc() ;
palette=[0 0 0 ; 0 0 1 ; 0 1 0 ; 1 0 0 ; 1 0.5 0 ; 1 1 0 ; 1 1 1]
xset("colormap",palette)
Black=1 ; Blue=2 ; Red=4 ; White=7 ;
D=100 ; square(0,0,D,D) ;
// Un cadre bleu, d'épaisseur 1
xset("pattern",Blue) ;
xfrect(0,D,D,D) ;
```

```
d=1 ;
xset("pattern",White) ;
xfrect(d,D-d,D-2*d,D-2*d) ;
// Préparation du rectangle
// Point de départ, hauteur, largeur
x=40 ; y=10 ; h=2 ; l=20 ;
xset("pattern",Red) ;
xfrect(x,y,l,h) ;
// déplacement vers le haut
delta=0.2
Delai=10 ; Stv=400 ;
for k=0 :Stv do
    y=y+delta ;
    xpause(Delai) ;
    xset("pattern",White) ;
    xfrect(x,y-h,l,delta) ;
    xset("pattern",Red) ;
    xfrect(x,y,l,delta) ;
end
```

La fonction fait appel à la primitive `xpause()`, qui permet d'interrompre le programme pendant une durée au moins égale, en millisecondes, à son paramètre.

1.4.3 Mise en place des éléments

Un listing complet est donné page 103. Comme toute application non triviale (c'est-à-dire ayant nécessité au moins quelques heures de travail), elle paraît bien compliquée a posteriori. La programmation a été réalisée de manière modulaire, en tenant compte des contraintes du langage pour permettre l'accès à certaines données communes à plusieurs sous-programmes. Le rôle dévolu aux diverses fonctions est le suivant :

han(W) Il s'agit de la fonction pricipale, la plus longue également, qui va valider son paramètre, initialiser le mode graphique, et enfin effectuer un appel à la fonction *hanoi()* qui implante l'algorithme effectif.

hanoi(D1,D2,D3,N,A,B) Cette fonction a pour objectif d'assurer le passage de N disques de A vers B. Comme il est nécessaire de dessiner les déplacements, il faut à chaque instant connaître la position des disques. Celle-ci est décrite par les variables *D1*, *D2* et *D3*, qui sont des vecteurs contenant, pour les piquets 1, 2 et 3 les numéros des disques présents. Pour une configuration à quatre disques, les valeurs initiales sont *[1,2,3,4]*, *[]* et *[]*.

hanmove(D,A,h1,B,h2) Déplacement du disque D de A (hauteur *h1*) vers B (hauteur *h2*)

hanmvup(A,D,h) Dessiner la montée du disque D situé à la hauteur h du piquet A.

hanmvh(A,B,D) Dessiner le déplacement horizontal du disque D, de A vers B.

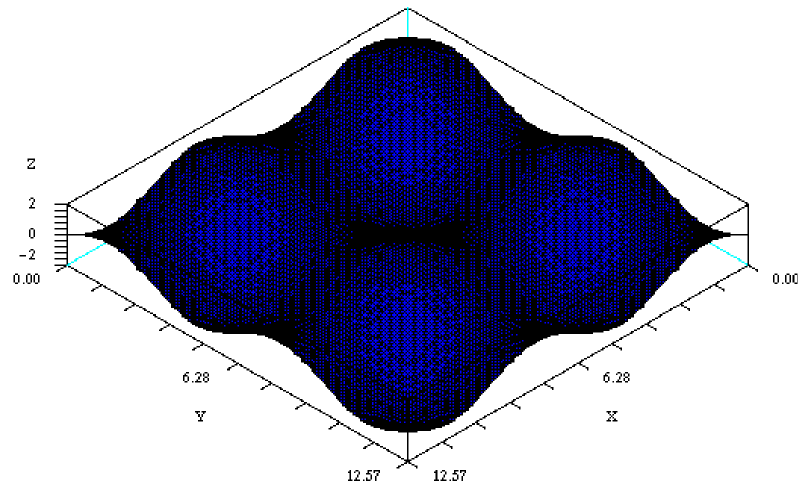

FIG. 1.5 – Fonction $z = \sin x + \sin y$

hanmvdown(B,D,h) Dessiner la descente du disque D sur le piquet B jusqu'à la position h.

hanboard() Tracé initial du support et des trois piquets.

hanmvdsk(D1,D2,D3,N,A,B) Fonction de mise à jour des configurations *D1*, *D2* et *D3* décrites ci-dessus, lorsque N disques sont déplacés de A vers B.

hanmvlast(D1,D2,D3,A,B) Fonction similaire à la précédente, pour un seul disque. Elle fournit également en résultat le numéro du disque déplacé, et sa position sur les piquets A (avant déplacement) et B (après déplacement).

1.5 Graphique 3D

Les fonctions graphiques que nous avons présentées jusqu'à présent opéraient dans le plan, c'est-à-dire en deux dimensions. Nous allons voir comment tracer des graphiques en trois dimensions.

Lorsque la surface à tracer a une équation du type $z = f(x, y)$, les opérations d'affichage sont remarquablement simples. Choisissons une fonction telle que : $z = \sin x + \sin y$. Il suffit de fournir à la fonction `plot3d()` trois paramètres, qui sont le vecteur des valeurs en x, comportant p éléments, le vecteur des valeurs en y, comportant q éléments, et la matrice, de taille $p \times q$ des valeurs correspondantes. Voici les instructions SCILAB :

```
-->x=linspace(0,4*%pi,129) ;
-->y=x ;
-->z=outer(add,sin(x),sin(y)) ;
-->size(z)
 ans =! 129. 129. !
-->plot3d(x,y,z)
```

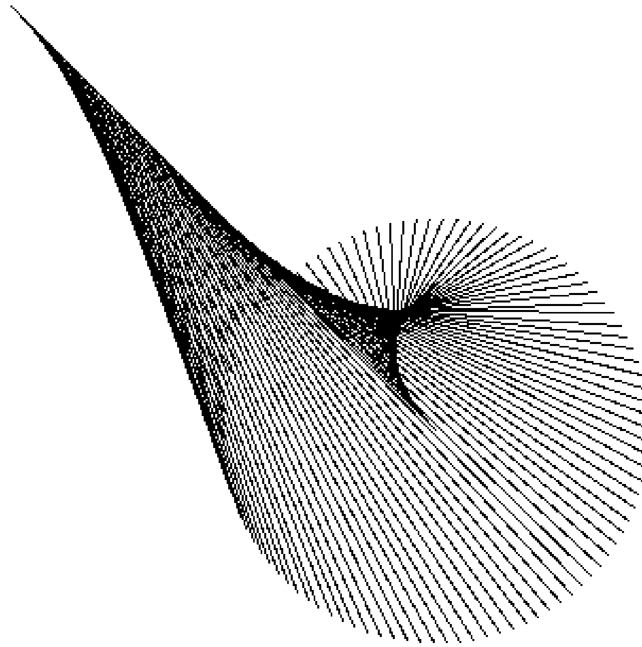

FIG. 1.6 – Tableau de fils

(nous faisons appel ici à la fonction `outer`, vue au § ??, page ??, et à la fonction `add`, qui ajoute ses deux opérandes)

Le résultat, visible sur la figure 1.5, page 19, représente la fonction en perspective isométrique.

Il s'agit naturellement de l'une des utilisations les plus simples de la fonction `plot3d`. On consultera la documentation pour savoir comment ajouter une légende, changer le point de vue, l'échelle, etc. La fonction `plot3d1`, similaire à celle-ci, utilise un dégradé de couleurs lié aux valeurs de z.

1.6 Exercices

1.6.1 Démonstration

Se faire jouer quelques unes des démonstrations graphiques accessibles par le menu DEMO de la fenêtre principale de SCILAB.

1.6.2 Tableau de fils

On se propose de réaliser un «tableau de fils», selon la mode des années 70, dans lequel on joint par des «fils» des points pris sur deux courbes géométriques. La figure 1.6, page 20, montre une telle figure, obtenue en joignant des points pris sur un cercle à des points pris sur une droite. Réalisez quelques figures dans ce style, si possible plaisantes à l'oeil.

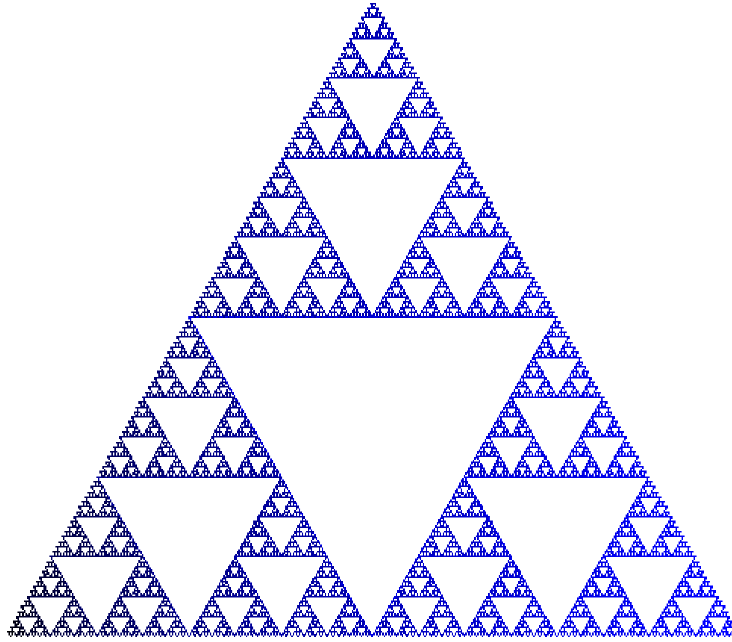

FIG. 1.7 – Triangle de Serpinsky d'ordre 8

1.6.3 Palettes graphiques

Écrire un programme qui affiche une palette graphique sous la forme d'une série de bandes (horizontales ou verticales). Concevoir un algorithme permettant de créer une palette graphique contenant les N dégradés entre deux couleurs données, avec une même intensité. Concevoir un algorithme permettant d'obtenir les N dégradés entre une une couleur donnée et le blanc, ou le noir.

1.6.4 Triangle de Serpinsky

Le triangle de Serpinsky, dont la figure 1.7 page 21 donne une représentation, est une fractale, dans laquelle des triangles sont dessinés à l'intérieur d'autres triangles. Plus précisément, on considère un triangle équilatéral $\widehat{ABC}$, de base BC, ainsi que les points A', B' et C', milieux des côtés BC, CA et AB respectivement. Alors :

- le triangle de Serpinsky d'ordre 1 basé sur $\widehat{ABC}$ est constitué du triangle $\widehat{A'B'C'}$;
- le triangle de Serpinsky d'ordre 2 basé sur $\widehat{ABC}$ est constitué de l'union du triangle de Serpinsky d'ordre 1 basé sur $\widehat{ABC}$ et des triangles de Serpinsky d'ordre 1 basés sur les triangles $\widehat{A'BC'}$, $\widehat{A'B'C}$ et $\widehat{AB'C'}$.
- et ainsi de suite, jusqu'à l'ordre N.

Écrire une fonction qui affiche le triangle de Serpinsky d'ordre N inscrit dans le cercle unité, en repère orthonormé.

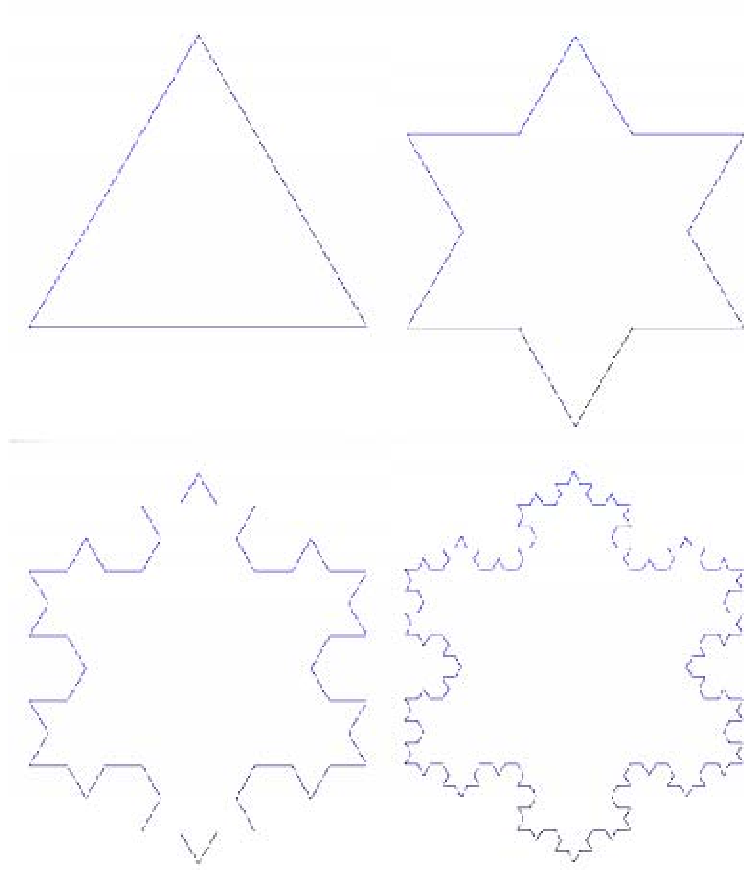

FIG. 1.8 – Flocon de Koch, pour N=0,1,2 et 3.

1.6.5 Flocon de Koch

Le flocon de Van Koch dont la figure 1.8 page 22 nous donne quatre étapes successives, est une courbe fractale plane basée sur un triangle équilatéral. Le flocon de Koch d'ordre N est obtenu de la manière suivante :

- chaque segment AB intervenant dans la figure (initialement un triangle équilatéral) est remplacé par les quatre segments AA', $A'C$, CB' et $B'B$, A' étant un point du segment AB situé au tiers du segment ($A'B = 2 \times AA'$), B' étant un point du segment AB situé aux deux tiers du segment ($AB' = 2 \times B'B$), et C étant le troisième sommet du triangle équilatéral $A'B'C$, tel que C soit situé à droite du segment AB.
- le paramètre N indique le nombre d'itérations à effectuer.

Écrire une procédure SCILAB dessinant le flocon de Koch d'ordre N.

1.6.6 Courbe du dragon

La courbe dite «du dragon» est une courbe fractale plane, obtenue de la manière suivante :

- la courbe du dragon d'ordre 1 entre deux points P et Q est le segment de droite joignant ces points ;
- la courbe du dragon d'ordre n entre deux points P et Q est constituée par l'union de la

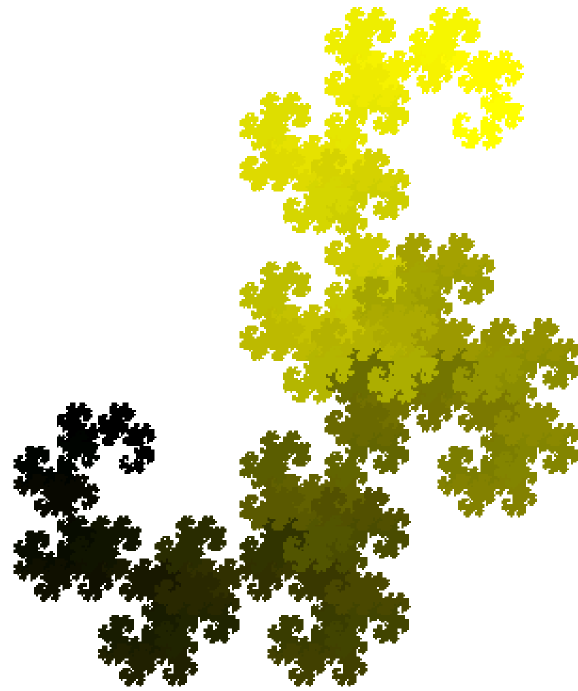

FIG. 1.9 – Courbe du Dragon

courbe du dragon d'ordre $n-1$ entre P et R, et de la courbe du dragon d'ordre $n-1$ entre Q et R, où R est le point tel que PQR est un triangle rectangle isocèle en R, R se trouvant à droite du segment PQ.

Écrire un programme `Dragon(n)` représentant, dans le carré [0,0,1,1] la courbe du dragon d'ordre n entre le point (0.3,0.4) et le point (0.7,0.8).

Combiner ce programme avec la création de palettes graphiques (présentée au § 1.6.3), afin d'obtenir un tracé de la courbe du dragon qui passe progressivement d'une couleur à une autre entre le point de départ et le point d'arrivée. La figure 1.9, page 23, donne une idée du résultat attendu.

1.6.7 La tortue Logo

Célèbre il y a bien longtemps, la *tortue Logo* était un mécanisme d'affichage graphique fort simple et fort astucieux, qui obéissait aux principes suivants :

- la tortue Logo se déplace sur un plan ; elle avance en ligne droite d'une distance donnée, ou tourne sur elle-même d'un angle donné ; elle est munie d'un crayon, qui peut être levé ou baissé. Lorsque le crayon est baissé, la tortue laisse une trace sur le plan en se déplaçant.
- les fonctions `penup()` et `pendown()` permettent de lever et de baisser le crayon ;
- la fonction `turn(A)` permet à la tortue de tourner sur elle-même d'un angle A ; *attention, les angles sont donnés en **degrés**.*
- la fonction `move(d)` permet à la tortue d'avancer d'une distance `d`.
- la tortue se déplace dans un repère orthonormé ; elle est initialement positionnée au point (0,0), elle est tournée vers les x positifs, et le crayon est levé. La fonction `tinit()` permet de se retrouver dans cette configuration.

Programmez les diverses fonctions décrites ci-dessus. Testez sur un exemple simple, tel que le tracé du carré :

```
tinit() ; pendown() ;
move(1) ; turn(90) ; move(1) ; turn(90) ;
move(1) ; turn(90) ; move(1) ;
```

En utilisant ces primitives, programmez le dessin du flocon de Koch.

Chapitre 2

Les chaînes de caractères

2.1 Présentation

Une *chaîne de caractères* est un objet du langage constitué d'une suite de caractères. On peut représenter dans une expression une chaîne de caractères sous la forme de la suite de ces caractères, placée entre apostrophes. Les caractères *apostrophe* ou *double-apostrophe*, dits parfois *quote* et *double-quote*, peuvent être utilisés pour délimiter les chaînes de caractères. Ces deux délimiteurs sont équivalents :

```
-->'Une chaîne de caractères'
 ans  =
 Une chaîne de caractères
-->"Une autre"
 ans  =
 Une autre
```

Notons qu'il faut doubler les apostrophes pour les représenter :

```
-->'L''été'
 ans  =
L'été
```

Certaines fonctions permettent de manipuler les chaînes de caractères :

```
-->s='Une chaîne de caractères'
 s   =
 Une chaîne de caractères
-->length(s)
 ans =
   24.
-->s+' plutôt longue'
 ans =
 Une chaîne de caractères plutôt longue
```

La fonction `length` appliquée à une chaîne fournit le nombre de caractères de celle-ci. Le dernier exemple montre la fonction de *concaténation* de chaînes de caractères, représentée par le symbole +.

La fonction `string` permet de convertir une valeur numérique en une chaîne de caractères. Le résultat comporte le nombre minimal de caractères permettant de représenter la valeur :

```
-->2+3
 ans  =  5.
-->string(2+3)
 ans  =  5
-->string(1/3)
 ans  =  0.3333333
```

Notons enfin qu'une chaîne de caractères est considérée comme un scalaire :

```
-->size("Bonjour")
ans =
! 1. 1. !
```

Tout comme les nombres, les chaînes de caractères peuvent être arrangées en vecteurs ou matrices. La fonction `emptystr()` permet de créer des tableaux de chaînes vides (tout comme `zeros()` permet de créer des matrices de zéros).

```
-->emptystr()
 ans =
-->''
 ans =
-->emptystr(2,5)
ans =
!            !
!            !
!            !
```

La notation utilisant crochets, virgules et point-virgules permet de créer des matrices contenant des chaînes non vides :

```
-->M=['une' 'matrice' ; 'de' 'caractères']
 M =
!une  matrice    !
!                !
!de   caractères !
```

On notera que la fonction `size()` fournit les caractéristiques de la matrice, alors que la fonction `length()` fournit les longueurs de chaînes de caractères de cette matrice :

```
-->size(M)
 ans =
! 2. 2.!
-->length(M)
ans =
! 3.  7. !
! 2.  10.!
```

2.2 Qu'est-ce qu'un caractère ?

Nos chaînes de caractères sont donc constituées de caractères. Mais qu'est-ce qu'un caractère ? Pour l'ordinateur, il s'agit simplement d'un code, un nombre choisi selon une convention locale ou universelle. Lorsque ce nombre est envoyé à une unité d'impression particulière (imprimante ou écran), le matériel affiche un dessin particulier (on parle de *glyphe*) représentant le caractère correspondant.

2.2.1 Les caractères en Scilab

Scilab utilise une convention qui lui est propre pour représenter les caractères. Il est possible de connaître cette représentation au moyen de la fonction `str2cod` (qui se prononce «str-to-code») :

```
-->str2code("abcd")
 ans =
 ! 10.!
 ! 11.!
 ! 12.!
 ! 13.!
```

Réciproquement, il est possible de synthétiser une chaîne de caractères, à partir d'un vecteur numérique contenant les codes des caractères que l'on désire obtenir :

```
-->code2str([12 10 11])
 ans = cab
```

De manière générale, on utilisera la représentation interne d'une chaîne de caractères chaque fois que l'on désirera manipuler, transformer cette chaîne. Ainsi, on remarquera que les lettres minuscules sont représentées par les entier consécutifs de 10 à 35, et que les lettres majuscules sont représentées par l'opposé de ces même codes :

```
-->str2code("azAZ")
ans =
!   10.!
!   35.!
! - 10.!
! - 35.!
```

On peut ainsi concevoir une fonction qui passe en majuscules la chaîne de caractères qui lui est transmise en paramètre :

```
-->deff("z=majuscules(s)", "z=code2str(-abs(str2code(s)))")
-->majuscules('Scilab')
ans = SCILAB
```

2.2.2 Autres représentations des caractères

La codification des caractères que nous venons de présenter, répétons le, est propre à Scilab. La codification la plus utilisée dans le monde informatique est l'ASCII (American Standard Codes for Information Interchange), dont le tableau 2.1, page 29, donne la représentation graphique, la codification, et le nom habituel. Le code ASCII comporte 128 caractères, numérotés de 0 à 127, dont 95 sont affichables. Ce code, qui ne comporte pas les lettres accentuées, ne convient pas aux langues indo-européennes. L'on utilise en Europe une extension du code ASCII, l'ISO-latin, qui comporte 256 caractères, les codes de numéros 192 à 255 contenant la plupart des caractères spécifiques des langues indo-européennes.

Scilab permet la conversion vers l'ASCII au moyen de la fonction `ascii()` : cette opération est utile lorsqu'il s'agit de manipuler des textes externes à Scilab (par exemple, lus dans un fichier). Elle transforme une chaîne de caractères en un vecteur de codes ASCII, et réciproquement.

```
-->ascii("abcd")
ans =
! 97. 98. 99. 100. !
-->ascii([65 66 67 97 98 99])
ans = ABCabc
```

2.3 Manipulation des chaînes de caractères

Plusieurs fonctions permettent la manipulation de chaînes de caractères.

2.3.1 Concaténation

Nous avons déjà vu que l'opération + permettait la *concaténation* de deux chaînes. Cette opération peut s'appliquer à des chaînes simples, mais aussi à des tableaux de chaînes, avec éventuellement une extension d'un opérande lorsque celui-ci est un scalaire :

```
-->"ab"+"cde"
 ans = abcde
-->["abc", "def", 'x', "yz"] + ['1' '2' '3' '4']
 ans =
!abc1  def2  x3  yz4  !
-->"AB" + ["abc", "def", "x", "yz"]
```

n°		nom	n°		nom	n°		nom
32		space	64	@	at	96	‘	grave
33	!	exclam	65	A	A	97	a	a
34	"	quotedbl	66	B	B	98	b	b
35	#	numbersign	67	C	C	99	c	c
36	$	dollar	68	D	D	100	d	d
37	%	percent	69	E	E	101	e	e
38	&	ampersand	70	F	F	102	f	f
39	’	apostrophe	71	G	G	103	g	g
40	(	perenleft	72	H	H	104	h	h
41	)	perenright	73	I	I	105	i	i
42	*	asterisk	74	J	J	106	j	j
43	+	plus	75	K	K	107	k	k
44	,	comma	76	L	L	108	l	l
45	-	bar	77	M	M	109	m	m
46	.	period	78	N	N	110	n	n
47	/	slash	79	O	O	111	o	o
48	0	0	80	P	P	112	p	p
49	1	1	81	Q	Q	113	q	q
50	2	2	82	R	R	114	r	r
51	3	3	83	S	S	115	s	s
52	4	4	84	T	T	116	t	t
53	5	5	85	U	U	117	u	u
54	6	6	86	V	V	118	v	v
55	7	7	87	W	W	119	w	w
56	8	8	88	X	X	120	x	x
57	9	9	89	Y	Y	121	y	y
58	:	colon	90	Z	Z	122	z	z
59	;	semicolon	91	[	bracketleft	123	{	braceleft
60	<	less	92	\	backslash	124	|	bar
61	=	equal	93	]	bracketright	125	}	braceright
62	>	greater	94	^	asciicircum	126	~	asciitilde
63	?	question	95	_	underscore	127		

FIG. 2.1 – Table des codes ASCII

```
 ans =
!ABabc  ABdef  ABx  AByz  !
```

La fonction `strcat()` permet la concaténation des chaînes contenues dans son opérande, qui doit être un tableau de chaînes :

```
-->strcat(["abc", "def", "x", "yz"])
 ans = abcdefxyz
-->strcat(["abc", "def" ; "x", "yz"])
 ans = abcxdefyz
```

On notera que la fonction prend en compte les éléments du tableau colonne par colonne. Lorsque `A` et `B` sont des chaînes simples, `A+B` équivaut à `strcat([A B])`.

La fonction `strcat()` peut être utilisée avec deux opérandes. Le second opérande est alors inséré entre chaque élément du tableau du premier opérande :

```
-->strcat(["abc", "def", "x", "yz"], "*")
 ans = abc*def*x*yz
```

2.3.2 Extraction de sous-chaîne

La fonction `part()` permet d'indexer une chaîne de caractères par un nombre ou un vecteur :

```
-->part("abcdefg", 2)
 ans = b
-->part("abcdefg", [2 3 4 6])
 ans = bcdf
```

Lorsque l'index utilisé est plus grand que le nombre d'éléments du tableau, le résultat est un caractère blanc :

```
-->part("ab", [1 3 5 2 4 2])
 ans = a  b b
```

Il est possible d'indexer des tableaux de chaînes ; le résultat est un tableau contenant le résultat de l'indexation individuelle de chaque élément du premier opérande par le second :

```
-->part(["abcdefg", "xyz"], [2 3 4 6])
 ans =
!bcdf  yz   !
```

2.3.3 Recherche de sous-chaîne

La fonction `strindex()` permet de rechercher une chaîne à l'intérieur d'une autre chaîne. La fonction fournit les indices de début de toutes les occurrences de la chaîne recherchée :

```
-->strindex("Mississippi", "ss")
 ans =! 3. 6. !
-->strindex("Mississippi", "i")
 ans =! 2. 5. 8. 11. !
```

La fonction permet la recherche simultanée de plusieurs sous-chaînes, que l'on fournit sous la forme d'un tableau de chaînes, comme second opérande :

```
-->strindex("Mississippi", ["i", "si"])
 ans =! 2. 4. 7. 11.!
```

On notera qu'il n'y a pas recouvrement des résultats : une nouvelle recherche débute après la fin de la sous-chaîne précédente.

2.3.4 Remplacement de chaînes

La fonction `strsubst()` permet de substituer une chaîne à une autre dans une chaîne ou un tableau de chaînes. Son premier paramètre est la chaîne ou le tableau de chaînes, le second la chaîne à remplacer, le troisième la chaîne de remplacement. La fonction effectue tous les remplacements possibles (c'est-à-dire aux emplacements qu'aurait trouvés `strindex()`).

```
-->S="à bon chat, bon rat"
 S = à bon chat, bon rat
-->strsubst(S, "at", "ien")
ans = à bon chien, bon rien
-->S
S = à bon chat, bon rat
```

Le résultat de la fonction est la chaîne modifiée ; le paramètre d'origine est inchangé.

2.3.5 Élimination de blancs

Dans le traitement des chaînes de caractères, il est souvent utile de supprimer les blancs excédentaires : blancs de début ou de fin, blancs redondants, etc. La fonction `strsubst()` permet naturellement de supprimer *tous* les blancs d'une chaînes :

```
-->a="   123  235    339  ";
-->strsubst(a, " ", "")
ans = 123235339
```

La fonction `stripblanks()` permet d'éliminer les blancs de *début* et de *fin* d'une chaîne, sans supprimer les blancs internes :

```
-->stripblanks(a)
 ans = 123  235    339
```

Enfin, le problème consistant à éliminer les blancs redondants peut se résoudre ainsi :

```
-->flag=%t ;
-->while flag do
-->    w = strsubst(a, '  ', ' ') ;
-->    flag = w <> a ;
-->    a = w ;
```

```
-->end
-->a
 a = 123 235 339
```

L'algorithme consiste tout simplement à remplacer toute suite de deux blancs par un seul.

2.3.6 Transformations majuscules/minuscules

Citons enfin la fonction `convstr()` qui permet, à l'instar de la fonction `majuscules()` définie au § 2.2.1, de changer la casse d'une chaîne de caractères :

```
-->convstr('Scilab')
 ans = scilab
-->convstr('Scilab', 'l')
 ans = scilab
-->convstr('Scilab', 'u')
 ans = SCILAB
```

Le second paramètre, optionnel, précise le passage en majuscules (`'u'`) ou en minuscules (`'l'`).

2.4 Divers

Décrivons encore quelques autres opérations, plus ou moins liées aux chaînes de caractères.

2.4.1 Lecture de caractères

La fonction `readc_()`[1] permet la lecture d'une chaîne de caractères. L'exécution du programme courant est suspendu jusqu'à ce qu'une chaîne de caractère (éventuellement vide), suivie d'une fin d'entrée (touche Enter ou Return) ait été tapée par l'utilisateur.

```
-->readc_()
-->w
ans = w
```

On peut utiliser cette opération pour réaliser des interactions avec l'utilisateur, au moyen d'une fonction telle que :

```
function Z=ques(S)
disp(S + ' (o/n)?');
c=readc_();
Z= c == 'o' | c == 'O';
```

qui permet par exemple le dialogue suivant :

```
-->ques('continuer les itérations')
 continuer les itérations (o/n)?
-->o
 ans = T
```

[1] Attention, le nom de la fonction se termine effectivement par le caractère «_».

2.4.2 Chaînes et paramètres

Une fonction qui attend comme paramètre une chaîne de caractères peut être utilisée sans les parenthèses qui marquent l'appel de fonction. L'objet qui suit est alors considéré comme une chaîne de caractères. Dans l'exemple suivant, la fonction `disp` a pour but d'imprimer son paramètre :

```
-->bonjour='hello'
 bonjour =
 hello
-->disp(bonjour)
 hello
-->disp bonjour
 bonjour
```

On notera que dans ce dernier cas, l'expression qui suit le nom de la fonction est considérée comme étant implicitement placée entre apostrophes, et n'est *pas* évaluée, contrairement à ce qui se passe lorsque l'on utilise des parenthèses :

```
-->disp 2+3
 2+3
-->disp(2+3)
 5.
```

2.5 Tableaux

Nous avons vu que, tout comme les nombres, les chaînes de caractères peuvent être assemblées en tableaux (vecteurs ou matrices). Les tableaux obtenus sont homogènes, en ce sens que tous les éléments doivent être des chaînes de caractères, mais leurs éléments n'ont pas nécessairement le même nombre de caractères :

```
-->A=['un','vecteur']
A =!un  vecteur  !
-->B=['un', 'tableau', 'de' ;'chaines','de','caractères']
B =
!un       tableau  de         !
!                             !
!chaines  de       caractères !
-->size(B)
ans =! 2. 3.!
-->length(B)
ans =
!  2.  7.  2. !
!  7.  2. 10. !
```

Quelles manipulations peut-on réaliser au moyen de tableaux de chaînes de caractères ? Certaines des fonctions décrites ci-dessus pour des vecteurs s'appliquent à des tableaux :

```
-->A'
ans =
!un       !
!         !
!vecteur  !
-->strcat(B," ")
ans = un chaines tableau de de caractères
-->strcat(B'," ")
ans = un tableau de chaines de caractères
```

2.6 Applications

On se propose d'écrire un programme nous permettant de trier une liste de mots. Nous écrirons pour ce faire un programme général de tri, et une fonction spécifique nous permettra de déterminer une relation d'ordre entre chaînes de caractères. La fonction utilisée pour le tri est la suivante :

```
function Z=bubble(V, cmp)
N=count(V) ;
Z=matrix(V,1,N) ;
encore=%t ;
while encore do
    encore=%f ;
    for k=1 :N-1 do
        if cmp(Z(k),Z(k+1)) then
            tmp=Z(k) ; Z(k)=Z(k+1) ; Z(k+1)=tmp ;
            encore=%t ;
        end ;
    end ;
end ;
```

Elle fait appel à un second paramètre, qui est la fonction permettant de comparer deux éléments du tableau transmis en paramètre. La comparaison de deux nombres pourrait s'effectuer avec une fonction telle que :

```
function R=cmpnum(A,B)
R=A<B ;
```

Voici un exemple de tri d'un vecteur numérique au moyen de cette fonction :

```
-->V=[2 9 7 8 3 2 0 5 3]
 V =! 2. 9. 7. 8. 3. 2. 0. 5. 3. !
-->bubble(V, cmpnum)
 ans =! 9. 8. 7. 5. 3. 3. 2. 2. 0. !
```

Comment réaliser la comparaison de deux chaînes ? Une première solution consiste à comparer les deux chaînes caractère par caractère, de la gauche vers la droite. La fonction `part()` nous permet d'extraire un caractère, la fonction `ascii()` sera utilisée pour obtenir le code associé au caractère. Voici une première version de l'opération :

```
function R=cmpstr(A,B)
// %t si A<B
N=max(length(A), length(B)) ;
for k=1 :N do
    a=ascii(part(A,k)) ;
    b=ascii(part(B,k)) ;
    if a<b then
        R=%t ; return ;
    end ;
    if a>b then
        R=%f ; return ;
    end ;
end ;
```

Un exemple de son utilisation :

```
-->M=['voici' 'une' 'phrase' 'dont' 'l''on' 'désire' 'trier' ...
--> 'les' 'mots' 'par' 'ordre' 'alphabétique' ] ;
-->bubble(M, cmpstr)
ans =
!voici une trier phrase par ordre mots les l'on dont désire alphabétique !
```

Bien entendu, notre fonction de comparaison est un peu frustre. On pourrait désirer par exemple qu'il y ait équivalence entre les lettres A, a, $\grave{a}$ et $\grave{A}$, que le tri se fasse dans l'ordre inverse, afin que les mots soient classés selon l'ordre alphabétique traditionnel, etc. Ces différents exercices sont laissés à la sagacité du lecteur.

2.7 Exercices

2.7.1 Chiffres Romains

Un grand classique est l'écriture en chiffres romains des nombres entiers. On se souviendra de la notation :

Lettre	Valeur
M	1000
D	500
C	100
L	50
X	10
V	5
I	1

On n'oubliera pas non plus que si 3 s'écrit *III*, 4 se note *IV*, 400 *CD*, etc. Exemples :

```
-->dec2rom(1924)
 ans = MCMXXIV
-->dec2rom(1949)
ans = MCMXLIX
```

Réciproquement, on se propose de réaliser la transformation inverse, c'est-à-dire la fonction qui décode une chaîne de caractères représentant un nombre écrit en chiffres romains, et fournissant l'entier représenté par cette notation.

```
-->rom2dec("MCMXXIV")
 ans = 1924.
```

2.7.2 Palindromes

Écrire une fonction reconnaissant si la chaîne de caractères qui lui a été passée comme paramètre est un palindrome, c'est-à-dire une phrase qui se lit de la même manière (à la ponctuation près), à l'endroit et à l'envers.

Quelques palindromes bien connus :
- Ésope reste ici et se repose.
- Élu par cette crapule.
- A man, a plan, a canal : Panama.

George Pérec (1936 - 1982) est l'auteur d'un palindrome de 1247 mots et plus de 76000 caractères, qui débute ainsi :

> Trace l'inégal palindrome. Neige. Bagatelle, dira Hercule. Le brut repentir, cet écrit né Pérec...

et se termine par :

> ... S'il porte, sépulcral, ce repentir, cet écrit ne perturbe le lucre : Haridelle, ta gabegie ne mord ni la plage ni l'écart.

2.7.3 Dictionnaire

On se propose de constituer un dictionnaire de la manière suivante :
- un fichier disque est fourni, qui contient un ensemble de mots de la langue française;

– on veut réaliser un programme qui lise cet ensemble de mots depuis le fichier disque, et les conserve dans une variable ou une structure quelconque ;
– on veut ensuite pouvoir manipuler cette structure, afin de savoir par exemple si un nom donné se trouve dans ce dictionnaire ; éventuellement, ce nom doit pouvoir être inséré dans le dictionnaire, s'il ne s'y trouve pas ;
– on désire enfin pouvoir enregistrer l'ensemble de ces mots, sous la forme d'un nouveau fichier texte, pouvant éventuellement être relu par la suite par notre programme.

2.7.4 Anagramme

On désire utiliser le dictionnaire construit à l'exercice précédent pour constituer un dictionnaire des anagrammes, c'est-à-dire un ensemble de classes d'équivalences dans lesquelles sont rangés les mots qui utilisent les mêmes lettres, mais pas dans le même ordre.

Écrire un programme qui lise le fichier décrit au § 2.7.3, et constitue ce dictionnaire des anagrammes.

Écrire une fonction qui fournisse tous les anagrammes d'un mot donné.

2.7.5 Conversions

On se propose d'écrire un programme qui prend comme entrée un entier positif ou nul, et imprime sa valeur sous forme de lettres, conformément à l'usage du français :

```
-->lettres(771)
ans = sept cent soixante et onze
-->lettres(791)
ans = sept cent quatre-vingt-onze
-->lettres(700)
ans = sept cents
```

Chapitre 3

Listes et matrices creuses

Ce chapitre est consacré à deux types de données particuliers, les listes et les matrices creuses.

3.1 Les listes : présentation

En Scilab, une *liste* est une suite d'éléments désignés par leurs indices. Le langage propose deux types de listes : les listes non typées, et les listes typées. Nous présenterons d'abord les listes non typées. Notons immédiatement que *les listes de* Scilab *n'ont rien à voir avec les structures de données habituellement désignées sous le nom de listes en informatique* : elles sont proches des *tableaux généralisés* de langages comme APL ou J.

3.1.1 Description des listes

Les listes sont une structure de données permettant d'assembler des objets de natures et de dimensions quelconques. Les éléments sont désignés par leurs indices, qui sont des entiers compris entre 1 et le nombre d'éléments de la liste.

3.1.2 Création de listes

Une liste est créée par l'opération `list`, et s'imprime sous la forme des valeurs de ses éléments :

```
-->l=list(1,2,'Hello',[1 2 3 4 5])
 l =
       l(1)
    1.
       l(2)
    2.
       l(3)
   Hello
       l(4)
 ! 1. 2. 3. 4. 5.!
```

3.2 Opérations sur listes

3.2.1 Taille

Les fonctions `length` et `size` permettent de connaître le nombre d'éléments d'une liste :

```
-->length(l)
 ans =  4.
-->size(l)
 ans =  4.
```

3.2.2 Indexation

Les éléments d'une liste peuvent être sélectionnés ou modifiés par l'opération d'indexation :

```
-->l(2)
 ans =  2.
-->l(4)(5)
 ans =  5.
-->l(4)(5)=100 ;
-->l(4)
 ans =
  ! 1. 2. 3. 4. 100. !
```

3.2.3 Addition et destruction d'éléments

L'opération d'indexation permet de créer ou de détruire des éléments. Si l'affectation fait référence à un indice supérieur au plus grand indice existant, la liste est augmentée d'autant d'éléments que nécessaire. Les éléments nouvellement introduits sont "non définis" :

```
-->l(8)=33
 l   =
       l(1)
     1.
       l(2)
     2.
       l(3)
 Hello
       l(4)
  ! 1. 2. 3. 4. 100. !
       l(5)
     Undefined
       l(6)
     Undefined
```

```
          l(7)
        Undefined
          l(8)
        33.
```

Un élément peut être supprimé d'une liste par affectation à cet élément de la *valeur nulle*, résultat de l'appel de la fonction `null()` :

```
-->l(6)= null() ;
-->l(1)= null() ;
-->l
 l =
       l(1)
    2.
       l(2)
 Hello
       l(3)
! 1. 2. 3. 4. 100.!
       l(4)
    Undefined
       l(5)
    Undefined
       l(6)
    33.
```

Dans cet exemple, les éléments 6, puis 1 ont été supprimés ; l'impression de la liste fait apparaître la nouvelle numérotation des autres éléments.

La *liste vide* peut être créée par l'appel de `list` sans paramètres :

```
-->l=list()
l = ()
```

Les notations `l(0)=x` et `l($+1)=x` permettent d'ajouter des éléments en tête et en queue de liste respectivement.

3.2.4 Extraction

Lorsque plusieurs éléments sont extraits d'une liste, il est nécessaire de prévoir autant de variables pour recevoir les résultats :

```
-->l=list(3,5,[],[3 4 5 9],'hello') ;
-->[A,B,C]=l(1 :3)
C = []
B = 5.
A = 3.
```

Opération	10	100	1000	10000
1 - Insertion en tête	0.364	11.191	925.00	96220
2 - Insertion en queue	0.433	16.230	1413.75	148660
3 - Insertion en queue	0.590	17.597	1426.25	149080
4 - Lecture d'élément	0.337	2.927	28.37	292
Lecture dans un vecteur	0.302	2.575	25.29	253
5 - Affectation d'élément	0.257	2.089	20.44	205

FIG. 3.1 – Performances des opérations sur listes

```
-->[A,B,C,D,E]=l( :) ;
-->D
D =! 3. 4. 5. 9. !
```

3.2.5 Concaténation de listes

La fonction `listcat()` permet de concaténer, c'est-à-dire mettre bout à bout, les listes qui lui sont passées en paramètre. Cette opération ne modifie pas les valeurs de ces paramètres :

```
-->l1=list(2,3,5) ;
-->l2=list() ;
-->l3=list("Hello", "world") ;
-->lstcat(l1,l2,l3)
ans =
ans(1) 2.
ans(2) 3.
ans(3) 5.
ans(4) Hello
ans(5) world
```

3.2.6 Note sur l'efficacité des diverses opérations

Le tableau 3.1, page 42, donne les résultats des mesures des expressions suivantes :

1. insertion en tête : `l(0)=k` ;
2. insertion en queue : `l(k)=k` ;
3. insertion en queue : `l($+1)=k` ;
4. lecture d'élément dans une liste : `w=l(k)` ; pour comparaison, des chiffres similaires sont fournis dans le cas de vecteurs ;
5. affectation d'élément dans une liste existante : `l(k)=k` ;

3.3 Matrices creuses

3.3.1 Présentationsont

Les *matrices creuses* (en anglais *sparse matrices*) constituent un autre type de données en SCILAB. Il s'agit ici de représenter des tableaux numériques dont un grand nombre d'éléments sont nuls. Il est possible, pour limiter l'encombrement en mémoire, de ne garder que les valeurs des éléments non nuls, ainsi que leurs positions dans la matrice. Ainsi, la matrice suivante contient 40 éléments, sont 5 seulement ne sont pas nuls :

```
-->eye(5,8)
ans =
! 1. 0. 0. 0. 0. 0. 0. 0. !
! 0. 1. 0. 0. 0. 0. 0. 0. !
! 0. 0. 1. 0. 0. 0. 0. 0. !
! 0. 0. 0. 1. 0. 0. 0. 0. !
! 0. 0. 0. 0. 1. 0. 0. 0. !
```

Sa représentation sous la forme d'une matrice creuse ne nécessite plus que 5 éléments, auxquels sont associés des entiers indiquant les positions dans la structure :

```
-->sp=sparse(eye(5,8))
sp =
( 5, 8) sparse matrix
( 1, 1) 1.
( 2, 2) 1.
( 3, 3) 1.
( 4, 4) 1.
( 5, 5) 1.
```

Les différentes opérations du langage s'appliquent à de telles représentations, et nombre d'entre elles conservent la structure «creuse» de ces objets :

```
-->size(sp)
ans =! 5. 8. !
-->size(sp')
ans =! 8. 5. !
-->sum(sp)
ans = 5.
-->mean(sp)
ans = 0.125
-->sp*sp'
ans =
( 5, 5) sparse matrix
( 1, 1) 1.
```

Opération	Description
`full(A)`	matrice pleine identique à `A`
`speye(n,p)`	matrice identité creuse de taille $n \times p$
`spones(A)`	matrice creuse de même taille que `A` avec des `1`
`sprand(n,p,d)`	matrice aléatoire de densité d, de taille $n \times p$
`spzeros(n,p)`	matrice creuse de zéros, de taille $n \times p$

FIG. 3.2 – Opérations de création de tableaux creux

```
( 2, 2) 1.
( 3, 3) 1.
( 4, 4) 1.
( 5, 5) 1.
-->sp(1,8)=3
sp =
( 5, 8) sparse matrix
( 1, 1) 1.
( 1, 8) 3.
( 2, 2) 1.
( 3, 3) 1.
( 4, 4) 1.
( 5, 5) 1.
```

3.3.2 Quelques opérations

On l'a noté, la plupart des opérations du langage vont s'appliquer, et pour certaines, préserver la structure de ces objets. Il existe par ailleurs des opérations spécifiques de création, qui vont permettre de définir des tableaux creux ayant certaines propriétés. Le tableau 3.2, page 44, décrit les opérations de création ou de manipulation de matrices creuses. Bien que leurs noms soient semblables à certaines opérations de création de matrices pleines, on notera que ces opérations ne sont pas toujours similaires à celles-ci. Ainsi, la fonction `spones()` n'est pas réellement une contrepartie de `ones()`, puisqu'elle ne positionne des `1` que dans les emplacements correspondant à des valeurs non nulles de son paramètre. De même, `sprand()` utilise un troisième paramètre, qui représente la densité de la matrice à créer.

On notera enfin que les exercices du chapitre proposent de réaliser quelques opérations complémentaires pouvant être utiles dans la manipulation des matrices creuses.

3.4 Exercices

3.4.1 Tableaux creux

On se propose de réaliser les opérations suivantes :

- l'opération `issparse()`, qui, appliqué à un objet, nous indique si celui-ci est un tableau creux ;
- l'opération `spcount()`, qui, appliquée à un tableau creux, nous fournit le nombre d'éléments non nuls de ce tableau ;
- l'opération `spdiag()`, qui, appliquée à un vecteur (creux ou non), nous fournit la matrice creuse dont la diagonale a pour éléments ceux du vecteur, et qui, appliquée à un tableau (creux ou non), nous fournit un vecteur creux représentant la diagonale de ce tableau.

Attention, les tableaux creux peuvent être de grande taille, et il n'est pas conseillé de passer pour ces opérations par un tableau plein intermédiaire !

Chapitre 4

À l'intérieur de Scilab

> Outside of SCILAB, a comics book is a scientist's best friend.
>
> Inside of SCILAB, it's too dark to read. *(p.c.c. Groucho Marx)*

Ce chapitre va nous permettre de présenter quelques caractéristiques de *l'implantation* de Scilab. Tout utilisateur un peu sérieux de ce logiciel se doit de connaître un minimum de détail techniques.

4.1 Types de données

4.1.1 Présentation

Il est utile de connaître les divers *types de données* manipulées par Scilab. Certaines fonctions permettent de connaître le type (ou la représentation) des données manipulées dans un programme, et, par exemple, de choisir un algorithme propre à ces données.

La fonction `type(x)` fournit un entier, qui représente le *type interne* de l'objet `x` . Le tableau 4.1, page 48, décrit les divers types internes de l'implantation de SCILAB.

4.1.2 Quelques types de données

Voici quelques-uns des types gérés par SCILAB, avec si nécessaire une courte description et quelques exemples.

4.1.2.1 Tableaux numériques

Les tableaux numériques, de type réel ou complexe, sont les objets que nous avons manipulés dans les six premiers chapitres de ce cours :

```
-->type(3)
ans = 1.
```

Numéro	Description du type
1	tableau numérique, de type réel ou complexe
2	tableau de polynômes
4	tableau booléen
5	tableau creux
8	tableau d'entiers, représentés sur 1, 2 ou 4 octets
10	tableau de chaînes de caractères
11	fonction au format source
13	fonction compilée
14	bibliothèque de fonctions
15	tableau généralisé, dit «liste»
16	tableau généralisé typé, dit «liste typée»
128	pointeur

FIG. 4.1 – Types internes de données

4.1.2.2 Tableau de polynômes

Les polynômes sont des représentations symboliques des polynômes mathématiques à une variable. La fonction `poly()`, dans l'exemple ci-dessous, permet de créer le polynôme en x dont les racines sont 2 et -1.

```
-->p=poly([2,-1],"x")
p =
            2
 - 2 - x + x
-->type(p)
ans = 2.
```

L'objet `p` est un polynôme simple ; il est possible de constituer des tableaux de polynômes :

```
-->[p, 2*p-1 ; 1+p*p, 1]
ans =
!            2                            2 !
! - 2 - x + x               - 5 - 2x + 2x  !
!                                           !
!              2    3   4                   !
!   5 + 4x - 3x - 2x + x       1            !
```

Quelques opérations ; détermination de la variable d'un polynôme, changement de variable, degré, coefficients et racines d'un polynôme :

```
-->varn(p)
ans = x
-->varn(p,'k')
ans =
```

```
             2
 - 2 - k + k
-->degree(p)
ans = 2.
-->coeff(p)
ans =! - 2. - 1. 1. !
-->roots(p)
ans =! - 1.!! 2.!
```

Tout un ensemble de fonctions (que l'on peut consulter au chapitre «Polynomial Calculations» de l'aide en ligne) permettent de manipuler symboliquement ces objets :

```
-->factors(p)
ans =
ans(1) 1 + x
ans(2) - 2 + x
```

Et, bien entendu, d'évaluer ces polynômes pour diverses valeurs de la variable :

```
-->p
p =
             2
 - 2 - x + x
-->horner(p,3)
ans = 4.
-->horner(p,3+%i)
ans = 3. + 5.i
```

4.1.2.3 Tableaux booléens

Nous avons déjà vu les propriétés de ces objets...

4.1.2.4 Tableaux creux

4.1.2.5 Tableaux d'entiers

4.1.2.6 Tableaux de chaînes de caractères

4.1.2.7 Fonctions au format source

4.1.2.8 Fonction compilée

4.1.2.9 Bibliothèque de fonctions

4.1.2.10 Tableaux généralisés, dits «listes»

4.1.2.11 Tableaux généralisés typés, dits «listes typées»

4.1.2.12 Pointeurs

4.2 Nombres aléatoires

Le langage propose la possibilité de générer des nombres aléatoire. L'opération de base est la fonction rand(), qui se décline en plusieurs syntaxes.

À la base de tout «générateur aléatoire», il y a en général[1] un algorithme bien déterministe. La génération de nombres aléatoires est un vieux problème de l'informatique, dont l'une des solutions traditionnelles repose sur l'algorithme suivant :

- le générateur est basé sur une graine S (*seed*, en anglais), qui sert de point de départ ;
- pour chaque demande d'un nombre «aléatoire», on calcule une nouvelle valeur de la «graine) par la formule $S = modulo(A \times S + B, C)$. Les générateurs diffèrent les uns des autres par les valeurs de A, B, C et de la graine initiale.
- un nombre aléatoire N compris entre 0 et 1 est obtenu par $N = S/C$.

4.3 Utilisation de programmes extérieurs

L'utilisateur de Scilab est parfois amené à faire appel à des procédures extérieures au langage. Il peut y avoir plusieurs motifs à ceci : il souhaite utiliser une procédure déjà existante, bien écrite et largement testée. La procédure peut être disponible également sous forme compilée uniquement. Une autre motivation est liée aux problèmes de performances. Considérons par exemple, la fonction suivante, qui calcule la fonction de *Fibonacci* d'un nombre donné :

```
function [r] = fib(n)
if n < 2 then
    r = n;
else
    r = fib(n-1)+fib(n-2) ;
```

[1]Je dis ici «en général», car il existe, outre les générateurs logiciels qui vont être décrits, des générateurs matériels, qui peuvent reposer par exemple sur des systèmes électroniques produisant des bruits blancs, d'où l'on extrait des signaux numériques, ou encore sur des dispositifs mécaniques, comme le «Lava Generator», qui utilise les célèbres lampes à bulles des années 70...

```
end
```

Le programme fonctionne correctement, et fournit les valeurs attendues :

```
-->fib(3)
 ans = 2.
-->fib(5)
 ans = 5.
-->fib(8)
 ans = 21.
-->fib(30)
 ans = 832040.
```

Pourtant, ce dernier appel nous semble particulièrement long, ce que confirme la fonction `chrono` :

```
-->chrono('fib(30)')
 105410 ms.
```

Plus de cent-cinq secondes pour obtenir ce dernier résultat ! La raison, naturellement, en est que nous utilisons un algorithme exponentiel, et que le calcul de $fib(n+1)$ prend le double du temps de celui de $fib(n)$!

Sans changer d'algorithme pour autant, nous pouvons écrire le programme équivalent dans un langage compilé (tel que *C* ou *Fortran*).

Le programme suivant est écrit en *C*. Il comporte un sous-programme, `fibbo`, qui calcule la fonction de *Fibonacci* de son paramètre, et une autre fonction, `fib1c`, qui assure la liaison avec avec Scilab :

```
#include "/users/chic/girardot/scilab-2.5/routines/stack-c.h"
static int fibbo(int w)
{
    if (w < 2)
        return w ;
    else
        return fibbo(w-1) + fibbo(w-2) ;
}
int fib1c(int * a, int * r)
{
    *r = fibbo(*a) ;
}
```

Voici une utilisation de ce programme :

```
-->unix('make cours/f/test.o')
 ans = 0.
-->link('cours/f/test.o', 'fib1c', 'C')
linking files cours/f/test.o to create a shared executable
```

```
shared archive loaded
Linking fib1c (in fact fib1c)
Link done
 ans = 2.
-->c=call('fib1c', 8, 1, 'i', 'out', [1,1], 2, 'i')
 c = 21.
```

La première commande a pour but de compiler le programme `test.c` (on parle de *programme source*). Le résultat de cette compilation est un *programme «objet»*, `test.o`, qui contient la traduction en langage de la machine des instructions du programme source.

La seconde commande, qui fait appel à la procédure `link()` va intégrer ces instructions au système SCILAB. Le premier paramètre est le nom du fichier à charger, le second est le nom du point d'entrée, le troisième indique que le programme est écrit en langage *C*. Cette dernière précision permet à SCILAB de respecter les conventions d'appel spécifiques aux programmes écrits en *C*.

La dernière commande, enfin, est un appel de la procédure qui vient d'être chargée, à travers une fonction spéciale de SCILAB, `call()`. Elle comporte, dans l'ordre :
- le nom de la procédure;
- pour chaque paramètre d'entrée du programme,
  - la valeur à transmettre;
  - le numéro du paramètre correspondant du programme C;
  - le type déclaré de ce paramètre, ici `'i'` comme «*integer*».
- la chaîne de caractère `'out'`, qui indique que la liste des paramètres d'entrée est terminée;
- pour chaque paramètre correspondant à un résultat du programme :
  - la dimension attendue de la matrice résultat, ici 1 par 1;
  - le numéro du paramètre correspondant du programme C;
  - le type déclaré de ce paramètre, ici `'i'` comme «*integer*».

Le test de notre procédure écrite en *C* montre qu'elle fournit les mêmes résultats que la fonction SCILAB, tout en étant bien plus performante :

```
-->c=call('fib1c', 30, 1, 'i', 'out', [1,1], 2, 'i')
 c = 832040.
-->chrono('call(''fib1c'', 30, 1, ''i'', ''out'', [1,1], 2, ''i'')')
 71.25 ms.
```

Le rapport des temps d'exécution, `105410/71.25`, est voisin de *1500*. Il nous rappelle que SCILAB n'est pas un modèle d'efficacité pour certains types de calculs...

On consultera naturellement les descriptions des opérations `link` et `call` pour en savoir un peu plus sur leur mode d'emploi.

4.4 Exercices

Deuxième partie

Annexes

Annexe A

Scilab, le bétisier

Cette partie est présente à titre purement informatif ; elle n'a pas son équivalent dans le cours dispensé oralement. Par ailleurs, elle décrit tout un ensemble de problèmes, d'erreurs ou de conceptions erronées (aussi bien chez les utilisateurs du langage que dans le langage lui-même, du reste) qui ne vont guère vous concerner, puisque vous avez lu avec attention le reste de ce document, et que vous appliquez avec la plus grande rigueur les précautions d'usage. Mais bon, les perles d'un bétisier, c'est un peu comme les histoires drôles, on a toujours envie de les faire partager aux autres...

A.1 Les plaisirs de la notation

A.1.1 Syntaxe

La syntaxe du langage est remarquablement peu intuitive[1]. Par exemple, si `A` est une variable ayant reçu la valeur `5`, laquelle de ces deux expressions est-elle autorisée ?

```
- - A
~ ~ A
```

Vous l'avez deviné, bien que `-A` permette d'obtenir l'opposé de `A`, il faut écrire `-(-A)` pour obtenir l'opposé de ce résultat. Par contre, l'opération `~` considère que tout nombre non nul représente la valeur "*vraie*". `~A` vaut donc `F`, et `~~A`, qui est autorisée, `T`. La morale est qu'il faut assez systématiquement utiliser des parenthèses pour être certain que tout s'effectue dans l'ordre attendu.

Les crochets et accolades sont parfois équivalents, par exemple lorsqu'il s'agit de créer des vecteurs ou des matrices :

```
-->[1 2 3 4 5]
 ans =! 1. 2. 3. 4. 5. !
-->{1 2 3 4 5}
 ans =! 1. 2. 3. 4. 5. !
```

et il est même possible de les mélanger au sein d'une même expression :

[1] J'entends par là qu'elle fait de son mieux pour déjouer l'intuition de l'utilisateur.

```
-->[1 2 3 4 5}
 ans =! 1. 2. 3. 4. 5.!
```

sauf pour créer le tableau vide, auquel cas les accolades ne sont pas autorisées :

```
-->[]
 ans = []
-->{}
   !--error 1
incorrect assignment
```

A.1.2 Transposition

L'apostrophe, ou quote, représente la transposition... à condition d'être accolée à son opérande. Un blanc entre celui-ci et l'apostrophe génère une erreur...

```
-->X2'
 ans =
! 1. 2. 3.!
! 4. 5. 6.!
! 7. 8. 9.!
-->X2 '
    !--error 40
waiting for end of command
```

A.1.3 Les expressions mystérieuses

Que font les expressions suivantes ?

```
-->(a=5)
--> :
-->!toto=3
```

A.1.4 Vecteurs

Une variable complexe s'imprime sous une forme telle que `3.+4.i`. Si l'on a défini, pour des raisons de compatibilité avec Matlab, une variable `i=%i`, on peut même écrire des vecteurs tels que :

```
[1+i, 3+4i]
```

Hélas, ce qui ressemble furieusement à un vecteur de deux complexes est en réalité strictement équivalent à la notation suivante :

```
[1+%i, 7, %i]
```

et représente donc un vecteur de trois nombres...

A.1.5 De l'équivalence des chaînes

Une chaîne de caractères se représente entre simples ou doubles quotes. Une chaîne vide se représente par zéro caractères, placées entre simples ou doubles quotes... Cette chaîne vide peut être utilisée dans diverses expressions :

```
-->'abc' + ''
 ans = abc
-->'' + 'def'
 ans = def
```

Mais pas partout :

```
-->disp('abc')
 abc
-->disp('')

-->disp 'abc'
 abc
-->disp ''
        !--error 31
incorrect string
```

A.2 Les joies des variables

A.2.1 Variables de contrôle de boucle

Les variables de contrôle de boucle sont des variables purement locales à la boucle. Elles ne sont pas utilisables à l'extérieur de celle-ci. Elles ont en outre la propriété de faire disparaître une autre variable de même nom :

```
-->n=3
 n =
     3.
-->for n=1 :6 do
--> n ;
-->end
-->n
  !--error  4
undefined variable : n
```

A.2.2 Une variable est-elle définie ou non ?

La fonction `isdef()` nous indique si une variable est définie ou non. Dans une fonction, on peut utiliser ceci pour donner une valeur par défaut à un paramètre non transmis, ou une variable globale non définie :

```
-->if ~isdef('toto') then toto=1 ; end
```

Mais naturellement, ça ne marche pas partout :

```
-->if ~isdef('sin') then sin=2 ; end
                             !--error 25
bad call to primitive :sin
```

Quant à savoir ce qu'est une variable...

```
-->(if)
   !--error 4
undefined variable : if
```

A.2.3 De l'effacement des variables

La commande `clear` permet de faire disparaître des variables. Un résultat similaire peut être obtenu en affectant à la variable le résultat de la fonction `null()` :

```
-->toto=3 ;
-->toto=null()
-->toto
  !--error 4
undefined variable : toto
```

A.3 Des opérations sur tableaux

A.3.1 Le tableau vide

Le *tableau vide* est une grande source de joie, dans la mesure où son comportement semble avoir été décidé au coup par coup pour chaque fonction où il est susceptible d'intervenir :

```
-->tab=[1 2 3 4 ; 5 6 7 8] ;
-->tab+[]
ans =
! 1. 2. 3. 4. !
! 5. 6. 7. 8. !
```

Dans ce premier exemple, `[]` se comporte comme un tableau de zéros de même taille que l'autre opérande.

```
-->tab*[]
 ans = []
-->[]*tab
 ans = []
-->tab.*[]
 ans = []
-->[].*tab
ans = []
```

Dans ce second exemple, `[]` agit comme élément absorbant vis à vis des multiplications. Il n'est pas accepté comme opérande dans d'autres opérations :

```
-->max(tab,[])
            !--error 45
null matrix (argument # 2)
```

Les opérations qui pratiquent une réduction implicite le traitent de manières fort diverses :

```
-->max([])
 ans = []
-->sum([])
 ans = 0.
-->prod([])
 ans = 1.
-->min([])
 ans = []
```

Alors que `sum` et `prod` fournissent l'élément neutre de l'opération (`+` et `.*`) qu'ils appliquent, `max` et `min` rendent le tableau vide lui-même ; on eut pu attendre un résultat plus significatif, comme $-\infty$ et $+\infty$, éléments neutres des fonctions *maximum* et *minimum* réciproquement.

Notons enfin que, grâce au tableau vide, l'expression `A(x)=y` peut faire disparaître l'élément `A(x)`, pour peu que la variable `y` aie la malchance d'être égale au tableau vide...

A.3.2 Création de vecteurs

Un «idiome» de Scilab pour créer un vecteur par ajout d'éléments est : `V=[V,k]`, dans lequel `k` est le nouvel élément que l'on désire ajouter au «bout» de `V`. Il est conseillé de n'utiliser cet idiome que :

- si on ne connaît pas a priori la taille du vecteur résultat ;
- et si le nombre d'éléments reste raisonnable (quelques milliers).

Considérons en effet la fonction suivante :

```
function V=ttt1(N)
V=[]
for k=1 :N do
    V=[V,k] ;
end
```

Cette fonction définit un vecteur par les additions successives d'éléments de valeurs `1` à `N`. Alors que la complexité de cette fonction semble être $O(n)$, il faut être conscient que l'opération `[V,k]` a pour effet de créer un nouveau vecteur, contenant une copie des éléments de `V`, plus la nouvelle valeur `k`, puis d'affecter le résultat à `V`.

Nous allons comparer les performances de cette fonction avec la suivante, qui réalise la même tâche en faisant appel à des procédures extérieures pour la création de vecteur, l'addition d'élément, et la récupération de résultat :

```
function V=ttt2(N)
vecinit()
for k=1 :N do
    vecelt(k) ;
end
V=vecvalue()
```

Le tableau suivant décrit les performances (en millisecondes) de ces deux procédures en fonction du nombre d'éléments :

paramètre	10	100	1000	8620	10000	100000
ttt1	0.1638794	1.0992432	16.894531	765.625	1008.125	530030
ttt2	1.2524414	8.8476562	86.015625	765.625	835.000	8370

Pour de petites valeurs de N, la fonction `ttt1` est plus efficace, mais à partir d'un certain nombre de résultats (vers 8620 éléments), la fonction `ttt2` devient plus performante. Or, les opérations effectuées par les fonctions auxiliaires utilisées semblent complexes :

```
function vecelt(value)
global vecval vecptr vecsiz
if vecptr>vecsiz then
    vecsiz=vecsiz*2 ;
    vecval=[vecval,vecval] ;
end
vecptr=vecptr+1 ;
vecval(vecptr)=value ;

function vecinit()
global vecval vecptr vecsiz
vecsiz=128
vecval=zeros(1,vecsiz)
vecptr=0

function Z=vecvalue()
global vecval vecptr vecsiz
Z=vecval(1 :vecptr)
```

Comment expliquer les meilleures performances de cette version ? Une fois encore, l'analyse de complexité va nous permettre de comprendre le phénomène.

La notation `[V,k]` a pour effet de créer un nouveau vecteur, comportant un élément de plus que le vecteur original. Ce nouveau vecteur est obtenu par une opération de copie en mémoire. La fonction `ttt1` qui crée un vecteur de N éléments par de telle recopies successives dissimule donc un algorithme en $O(n^2)$.

Nos trois fonctions utilitaires, en revanche, vont gérer un vecteur dont la taille va s'accroître non pas à chaque insertion d'élément, mais chaque fois que la taille limite du vecteur est atteinte.

On crée alors un nouveau vecteur de taille double, par recopie. Le vecteur atteint donc la taille de N éléments par des doublement successifs, et le nombre d'étapes est donc en $O(\ln n)$. La fonction `ttt2` opère donc en $O(n \ln n)$, et il est donc explicable que ses performances deviennent meilleures pour une certaine valeur de N.

A.4 De la nature des fonctions

Les fonctions du langage ne sont pas toutes égales entre elles. Il y a les fonctions primitives du système, les fonctions prédéfinies du système, et enfin les fonctions définies par l'utilisateur. On peut faire certaines choses avec les fonctions définies, que l'on ne peut pas faire avec les fonctions primitives.

A.4.1 Leur création

Le but d'un langage de programmation semble être de définir des fonctions. Quoi de plus naturel que d'effectuer quelques tests en mode terminal ? Hélas, SCILAB ne nous y autorise pas :

```
-->function x=f(y)
   !--error 4
undefined variable : function
```

Il faut passer par la création d'un fichier, ou l'utilisation, malcommode, de `deff`.

A.4.2 De l'inégalité des fonctions

Il est possible d'imprimer l'en-tête d'une fonction définie, simplement en entrant son nom.

```
-->pgcd
 pgcd = [r]=pgcd(u,v)
```

Il est même permis d'affecter la valeur de cette fonction à une variable. Elle devient alors équivalente à la fonction.

```
-->f=pgcd
 f = [r]=f(u,v)
-->f(6,12)
 ans =  6.
```

On peut également passer cette valeur à une autre fonction :

```
-->vectorize(fact,1 :8)
 ans = ! 1. 2. 6. 24. 120. 720. 5040. 40320. !
```

Malheureusement, on ne peut pas faire de même avec les fonctions primitives :

```
-->sin
    !--error 25
bad call to primitive :sin
```

```
-->g=sin
   !--error 25
bad call to primitive :sin
-->vectorize(sin,1 :8)
   !--error 25
bad call to primitive :sin
```

Il faut cette fois créer explicitement une fonction définie équivalente, faisant appel à la fonction primitive :

```
-->deff('z=sinus(x)', 'z=sin(x)')
-->sin(0.123)
 ans =  0.1226901
-->sinus(0.123)
 ans =  0.1226901
-->vectorize(sinus,1 :6)
 ans =
! 0.8414710 0.9092974 0.1411200 - 0.7568025 - 0.9589243 - 0.2794155 !
```

De manière générale, il est nécessaire d'utiliser cette technique pour "passer" une fonction primitive, ou un opérateur, comme paramètre à une autre fonction.

A.4.3 De l'affectation des fonctions

Il est possible d'affecter à une variable la valeur d'une fonction définie, mais ce n'est en général pas une très bonne idée. La session ci-dessous devrait vous en convaincre :

```
-->getf('cours/f/fact.sci')
-->fact(10)
 ans =  3628800.
-->getf('cours/f/fib.sci')
-->fib(10)
 ans =  55.
-->fun=fib
 fun = [r]=fun(n)
-->fun(10)
 ans =  55.
-->fib=fact
Warning :redefining function : fib
fib = [a]=fib(n)
-->fib(10)
 ans =  3628800.
-->fun(10)
 ans =  403200.
```

Dans cet exemple, nous faisons appel à deux grands classiques, *factorielle* et *Fibonacci.* Après les avoir «testées» (plus que sommairement, nous tentons d'affecter à `fun` la valeur de `fib`, puis à `fib` la valeur de `fact`. Hélas, après tout ce méli-mélo, `fun` ne semble plus se comporter ni comme `fact`, ni comme `fib`. Que se passe-t-il ?

Annexe B

Principales commandes de Scilab

Cette annexe rappelle les principales commandes du logiciel, en donnant éventuellement la référence du paragraphe du cours dans lequel cette commande est décrite. On se reportera naturellement à la commande `help` pour toute précision supplémentaire.

B.1 Information

`help`	??	Aide en ligne : sans paramètre, ouvre une fenêtre décrivant la commande `help`. Suivi du nom d'une fonction du langage, ouvre une fenêtre décrivant cette fonction.
`apropos`	??	Aide en ligne : suivi d'un mot ou d'une chaîne de caractères, ouvre une fenêtre contenant un bref résumé des aides en ligne contenant ces mots.
`what`		Liste des mots-clefs du langage

B.2 Gestion de l'espace de travail

`who`	??	Liste des variables définies
`clear`		Avec des paramètres, efface les variables correspondantes
`global`	??	Déclaration de variable globale
`clearglobal`		Effacement de variable globale
`edit`		Appel d'un éditeur de texte (créer ou de modifier des fichiers de script)
`quit`	??	Arrêt de SCILAB

B.3 Contrôle divers

B.4 Mise au point

B.5 Variables du système

`MSDOS`	Variable booléenne, indiquant si l'on est sous MSDOS
`PWD`	Répertoire courant
`home`	Répertoire d'accueil de l'utilisateur
`TMPDIR`	Répertoire de travail temporaire de SCILAB
`SCI`	Répertoire où se trouve le logiciel SCILAB

Annexe C

Solution de quelques exercices

C.1 Exercices du chapitre 1

C.1.1 Hypoténuse

L'exercice consiste à calculer $\sqrt{A^2 + B^2}$. La documentation, consultée par `apropos square`, nous conseille la fonction `sqrt`. L'expression suivante résout le problème :

```
-->A=12 ; B=16 ;
-->sqrt(A^2+B^2)
 ans   =
    20.
```

On notera que l'on peut tout aussi bien écrire : `sqrt(A*A+B*B)`.

C.1.2 Disque

En exprimant les dimensions du disque en mètres et en mètres carrés (les unités légales), les deux formules équivalentes $\pi \times R^2$ ou encore $\frac{\pi \times D^2}{4}$ nous donnent :

```
-->D=0.12
 D =
     0.12
-->pi*(D/2)^2
 ans =
     0.0113097
-->pi*D^2/4
 ans =
     0.0113097
```

C.1.3 Nombres complexes

La documentation `apropos complex` ne nous fournit pas un résultat immédiatement exploitable ; par contre, `apropos imaginary` nous propose l'entrée `imag`, qui est exactement ce que

nous cherchons. De plus, la documentation de cette fonction nous renvoie à `real`. Il ne reste plus qu'à vérifier la chose :

```
-->C=3+4*i
 C    =
    3. + 4.i
-->real(C)
 ans  =
    3.
-->imag(C)
 ans  =
    4.
```

Pour calculer le module, on peut désormais, comme dans l'exercice sur l'hypoténuse (exercice C.1.1), calculer `sqrt(real(C)^2+imag(C)^2)`. On peut aussi essayer de trouver la fonction adéquate. La recherche sur "*modulus*" ne donne rien ; par contre, dans les résultats de la recherche sur "*magnitude*", nous trouvons `abs` qui répond à la question.

```
-->mag = abs(C)
 mag =
    5.
```

En ce qui concerne l'*argument*, ou la *phase*, il faut je le crains passer par l'*arc tangente* (on consulte `apropos tangent`, bien sûr) :

```
-->phi=atan(imag(C),real(C))
 phi =
     0.9272952
```

On notera que cette opération, avec deux paramètres, donne en fait la *phase*, qui est bien ce que nous recherchions. La fonction *arc tangente* serait obtenue en fournissant à la fonction un argument unique, qui serait le rapport des deux valeurs :

```
-->atan(imag(C)/real(C))
 ans =
     0.9272952
```

Le résultat est ici identique, mais aurait pu être différent pour d'autres valeurs du complexe :

```
-->U= -3+4*%i
 U =
     - 3. + 4.i
-->atan(imag(U),real(U))
 ans =
     2.2142974
-->atan(imag(U)/real(U))
 ans =
     - 0.9272952
```

L'opération inverse passe par le calcul des *sinus* et *cosinus* de l'angle :

```
-->mag*(cos(phi)+i*sin(phi))
 ans =
     3. + 4.i
```

Une écriture équivalente utilise $e^{i \times phi}$:

```
-->mag*exp(i*phi)
 ans =
     3. + 4.i
```

C.1.4 Affichage

Voici la trace des manipulations proposées :

```
-->N=1/3
 N   =
     0.3333333
-->0.3333333*3
 ans =
     0.9999999
-->N*3
 ans =
     1.
```

Le résultat nous est fourni avec 7 chiffres décimaux. La première multiplication semble nous donner un résultat cohérent (7 chiffres également) ; la seconde nous montre cependant que `0.3333333` n'est qu'une approximation de la valeur effective de `N`. En l'occurrence, SCILAB n'affiche les résultats qu'avec 7 chiffres significatifs, alors que la valeur est conservée avec une précision bien supérieure. L'instruction `format` permet de modifier le nombre de chiffres affichés :

```
-->format(16)
-->N
 N =
     0.3333333333333
-->format(30)
-->N
 N =
     0.3333333333333333314829616256
```

Le dernier exemple nous montre les limites de la représentation des nombres sur la machine. Le nombre N est représenté de manière exacte avec 16 chiffres significatifs seulement. Les autres décimales ne sont qu'un artefact dû à la procédure d'impression.

C.1.5 Approximations

Voici la trace des expressions :

```
-->X=1E30 ; Y=1E10 ;
-->X+Y-X
 ans =
     0.
-->X-X+Y
 ans =
     10000000000.
```

Si le second résultat correspond bien à ce que nous attendions, le premier nous laisse perplexe. Il nous faut nous souvenir que nous sommes face à une machine, dont la précision des calculs (on l'a vu dans l'exercice précédent) est limitée à 16 décimales environ. Or, le rapport `X/Y` est de 10^{20}, donc très supérieur à cette précision. Dans le premier exemple, l'addition est effectuée en premier, donnant un résultat identique à `X`. La soustraction qui suit fournit un résultat nul. Dans le second exemple, `X-X` est calculé en premier, fournissant le résultat 0 ; l'addition qui suit fournit la valeur `Y`.

C.1.6 Tracé de courbes

La demande "`help plot`" qui vient à l'esprit nous imprime la page décrivant la fonction `plot`. Il se trouve que le premier exemple d'utilisation de cette fonction correspond précisément à ce qui est demandé. Il ne reste plus qu'à taper (ou, moins fatigant encore, copier-coller) l'exemple dans la fenêtre Scilab :

```
--> x=0 :0.1 :2*%pi ;
--> // simple plot
--> plot(sin(x))
```

Il n'est même pas besoin de savoir ce que ça peut vouloir signifier pour pouvoir s'en servir...

C.1.7 Interface

Le point d'exclamation, nous apprend-on, rappelle une ligne de l'historique. Le cours nous dit aussi qu'un nom de variable peut débuter par un point d'exclamation. Si j'essaye d'écrire :

```
--> !toto=5
```

l'écran clignote, et le spot refuse de passer à la ligne suivante. Ceci nous indique simplement qu'il n'existe pas, dans l'historique, de ligne qui débute par "toto=5". Si une telle ligne avait existé, comme après :

```
-->toto=543
 toto = 543
```

le système nous aurait obstinément affiché "toto = 543". La solution consiste à mettre un blanc devant le " !" :

```
-->!toto=5
 !toto = 5.
```

Nous pouvons dès lors utiliser cette variable, toujours en prenant soin de ne pas faire débuter son nom en première colonne...

```
-->!toto+toto
 ans = 548.
```

C.1.8 Programme nu

Ceci est pour les aficionados d'UNIX ou de Linux...

La commande "`help scilab`" nous renseigne sur la possibilité de lancer SCILAB sans ouvrir la fenêtre d'interaction : l'option de lancement `-nw` indique à SCILAB de travailler directement dans la fenêtre du terminal d'où il est lancé.

Ceci est particulièrement pratique lorsque l'on veut utiliser le logiciel comme moteur de calcul. En demandant à SCILAB de lire les entrées sur un fichier, d'effectuer ses sorties sur un autre, on peut l'intégrer dans une chaîne de traitement plus complexe, lui faire exécuter automatiquement une série de calculs à des moments précis, etc...

C.1.9 Menus

La commande "`apropos menu`" nous indique une liste d'opérations permettant de gérer les menus sous SCILAB. La commande `addmenu` semble être un bon candidat pour résoudre le problème qui nous préoccupe. Il suffit de recopier l'un des exemples donnés, en le modifiant très légèrement :

```
-->addmenu('Salut',['Bonjour';'Au Revoir'])
-->Salut=['disp(''Hello'')';'disp(''Good Bye'')']
 Salut =
!disp('Hello')     !
!                  !
!disp('Good Bye')  !
```

On voit dès lors apparaître le nouveau menu, SALUT, dans la barre, et les sous-menus HELLO et GOOD BYE sont opérationnels.

C.1.10 Compatibilité

Comme signalé dans le cours, le fichier `.scilab`, lorsqu'il est placé dans le répertoire d'accueil de l'utilisateur, est automatiquement exécuté au lancement du logiciel. Ce fichier consiste en une série de commandes. Il est possible en particulier d'y placer des instructions telles que :

```
pi = %pi
i = %i
eps = %eps
nan = %nan
```

```
inf = %inf
t = %t
f = %f
T = %t
F = %f
```

qui permettront à l'utilisateur de disposer des variables `pi`, `i`, `eps`, etc., dès le lancement du logiciel.

C.2 Exercices du chapitre 2

C.2.1 Éléments de syntaxe

Le premier exemple est très certainement une erreur de syntaxe. L'instruction *if* attend une suite d'instructions dans les branches *then* ou *else* ; or `k=3` est une expression, l'instruction devant se terminer par une virgule, un point-virgule ou un passage à la ligne. Voici l'erreur en situation :

```
-->if 2==2 then
-->k=3 end
  !--error 40 waiting for end of command
```

La seconde expression est correcte ; il s'agit de la fin d'une instruction de type *if*, suivie d'une instruction d'affectation :

```
-->if 2==2 then k=1 ;
-->end k=3 ;
-->k
 k = 3.
```

La troisième expression peut également apparaître dans un contexte valide :

```
-->if 2==2 then
-->  if 3==3 then
-->    k=5 ;
-->end end
-->k
 k = 5.
```

La dernière expression est une partie *else* contenant une instruction vide, comme dans :

```
-->if 3==5 then k=3
-->else ; end
```

C.2.2 Quelques calculs

Voici la trace d'une session au cours de laquelle les différentes expressions sont traduites en Scilab :

```
-->3+8/5-7*4
 ans = - 23.4
-->(3^2-8)/2^2+3^2/5-6*11^2/7^3
 ans = - 0.0666181
-->exp(log(5))*(1-0.12^2)^(1/2)
 ans = 4.9638695
-->exp(log(5))*sqrt(1-0.12^2)
 ans = 4.9638695
-->cosh(1.5)^2-sinh(1.5)^2
 ans = 1.
-->sin(pi/6+acos(0.5))
 ans = 1.
-->a=4.770796326794897 ;
-->b=11.05398163397448 ;
-->cos(a)^2+sin(b)^2
 ans = 1.
```

Exponentielle : la fonction suivante calcule les $n+1$ premiers termes de la suite en n'effectuant qu'une multiplication et une division pour chaque terme, et est donc optimale en temps de calcul :

```
function [z]=exp1(x, n)
z=1
s=1
for k=1 :n do
   s=s*x/k ;
   z=z+s ;
end
```

Quelques exemples d'utilisation :

```
-->exp1(1.25, 10)
 ans = 3.4903426
-->exp(1.25)
 ans = 3.490343
-->exp1(3.31,20)
 ans = 27.385125
-->exp(3.31)
 ans = 27.385125
```

C.2.3 Nombres premiers

Un nombre est premier s'il n'est divisible que par lui-même et l'unité. Un algorithme consiste donc à tester les restes des divisions du nombre par différents entiers. La première solution, qui consiste à essayer tous les entiers compris entre 2 et $N-2$ n'est pas nécessairement la meilleure. Voici l'écriture d'une telle solution. Le résultat est initialement indiqué comme «vrai», mais si l'une des divisions fournit un reste égal à 0, la valeur «faux» sera affectée au résultat.

```
function [v]=premier0(n)
v=%t ;
for k=2 :n-1 do
    if modulo(n,k)==0 then v=%f ; end
end
```

Pour améliorer les performances du programme, on peut remarquer que :
- si le nombre n est divisible par 2, hormis 2 lui-même, il ne sera pas premier ;
- dans le cas contraire, l'on peut donc se passer de tester tous les diviseurs pairs ;
- enfin, que le nombre n ne sera pas divisible par k compris entre $n/2$ et n. On peut pousser plus loin cette réflexion, et se dire que si n n'est divisible par aucun des entiers compris entre 1 et $\sqrt{n}$, il ne sera pas divisible par un nombre k compris entre $\sqrt{n}$ et n.

Voici une solution possible, qui tente de limiter les calculs en itérant sur les seuls nombres impairs, et en s'arrêtant à la racine carrée du nombre :

```
function [v]=premier1(n)
v=%f
if modulo(n,2)==0 then return ; end
for k=3 :2 :ceil(sqrt(n)) do
  if modulo(n,k)==0 then return ; end
end
v=%t
```

Un exemple d'utilisation :

```
-->premier1(31)
 ans = T
-->premier1(99)
 ans = F
-->premier1(31991)
 ans = T
-->chrono('premier1(31) ;')
 0.4858398 ms.
-->chrono('premier1(31991) ;')
 8.2324219 ms.
-->chrono('premier0(31991) ;')
 2665 ms.
```

On peut naturellement envisager d'autres solutions, depuis la conservation de liste de nombres premiers jusqu'à des méthodes mathématiques beaucoup plus complexes, que l'on réserve en général aux grands nombres.

C.2.4 Nombres parfaits

Un nombre est parfait s'il est égal à la somme de ses diviseurs (hormis lui-même). On peut noter, comme on l'a fait remarquer dans l'exercice ci-dessus, qu'un entier p n'a pas de diviseurs compris entre $p/2$ et p. La fonction suivante répond au problème. Elle teste tous les diviseurs “possibles” du nombre, de *2* jusqu'à *p/2*.

```
function [r]=parfait(p)
k=1
for n=2 :floor(p/2) do
    if modulo(p,n)==0 then
        k=k+n ;
    end
end
r = k==p
```

Quelques nombres parfaits :

```
-->for w=1 :10000 do if parfait(w) then disp(w) ; end end
 1.
 6.
 28.
 496.
 8128.
```

On notera que cette boucle est très lente à exécuter : le test, `parfait`, est appliqué à tous les nombres de 1 à 10000 ; or ce test nécessite lui-même une boucle de 2 jusqu'à la moitié de la valeur du nombre. Le test central va donc être exécuté près de 10000×5000 fois, soit 50 millions de fois !

On peut améliorer l'algorithme, en remarquant que, si q est l'un des diviseurs de p, le nombre p/q sera aussi un diviseur de p ; que si q est le plus petit des deux nombres q et p/q, on aura forcément $q < \sqrt{p}$ et $p/q > \sqrt{p}$. Il suffit donc de chercher tous les diviseurs q de p compris entre 2 et $\lfloor \sqrt{p} \rfloor$, liste à laquelle on ajoutera tous les p/q.

La fonction suivante implémente cet algorithme :

```
function [r]=parfait2(p)
k=1
for n=2 :floor(sqrt(p)) do
    if modulo(p,n)==0 then
        k=k+n ;
        if (n <> p/n) then
```

```
            k=k+p/n ;
        end
    end
end
r= k==p
```

On peut vérifier que ses performances sont meilleures que la version précédente :

```
-->chrono('for k=1 :1000 do parfait(k) ; end')
 21210 ms.
-->chrono('for k=1 :1000 do parfait2(k) ; end')
 2043.75 ms.
```

La notion de «nombre parfait» remonte à Euclide, qui conjecture, dans ses éléments de géométrie, que si $p = \sum_{k=0}^{k=n} 2^k$ est premier, le produit $p * 2^n$ est un nombre parfait. Il mentionne les quatre premiers nombres parfaits, 6, 28, 496 et 8128. On notera que le cinquième nombre parfait connu, 33550336, est hors de la portée de nos fonctions...

C.2.5 PGCD

La fonction PGCD est relativement simple à mettre au point. Voici une première version, la plus simple possible, qui respecte le sujet original :

```
function A = pgcd(A,B)
while (A <> B)
    if A > B then
        A = A - B ;
    else
        B = B - A ;
    end
end
```

La version suivante incorpore quelques tests :

```
function r = pgcd(u,v)
// PGCD de u et v
if int(u) <> u | real(u) <> u then
    error("paramètre incorrect "+string(u)) ;
end
if int(v) <> v | real(v) <> v then
    error("paramètre incorrect "+string(v)) ;
end
if u < 0 then u = -u ; end
if v < 0 then v = -v ; end
if u == 0 | v == 0 then
```

```
        r = u+v
    else
        while (u <> v) do
            if u>v then
                z=u ; u=v ; v=z ;
            end
            v = v-u ;
        end
        r=u
    end
```

Quelques exemples :

```
-->pgcd(1547, 2431)
 ans = 221.
-->pgcd(2351, 7507)
 ans = 1.
 -->pgcd(2.17, 12)
    !--error 9999 paramètre incorrect 2.17
at line 4 of function pgcd
called by : pgcd(2.17, 12)
```

Pour aller plus loin : la fonction `type(x)` rend 1 si son paramètre est un tableau numérique.

C.3 Exercices du chapitre 3

C.3.1 Complexités et puissances des machines

On se place dans l'hypothèse simplificatrice où le temps T nécessaire à un programme P est proportionnel à la durée D d'une instruction élémentaire et à une fonction $f(N)$ du nombre d'éléments à traiter : $T = D \times f(N)$. Pour une autre machine, dont la durée d'une instruction est D', le nombre d'éléments traités sera N', avec $T = D' \times f(N')$. Le rapport de puissance, 1000, entre les deux machines nous permet d'écrire : $f(N') = 1000 \times f(N)$. L'application aux divers algorithmes nous donne, en prenant N=1000 éléments :

Complexité	Méthode de calcul	Nombre d'éléments
$O(\ln n)$	$N' = N^{1000}$	$N' \simeq +\infty$
$O(n)$	$N' = 1000 \times N$	$N' \simeq 1000000$
$O(n \ln n)$	$N' \times \ln N' = 1000 \times N \times \ln N$	$N' \simeq 144764$
$O(n^2)$	$N' =^2 \sqrt{1000} \times N$	$N' \simeq 31622$
$O(n^3)$	$N' =^3 \sqrt{1000} \times N$	$N' \simeq 10000$
$O(n^5)$	$N' =^5 \sqrt{1000} \times N$	$N' \simeq 3981$
$O(2^n)$	$N' = N + \frac{\ln 1000}{\ln 2}$	$N' \simeq 1010$
$O(3^n)$	$N' = N + \frac{\ln 1000}{\ln 3}$	$N' \simeq 1006$

On comprend mieux, en regardant les dernières lignes du tableau, pourquoi les algorithmes exponentiels déplaisent tant aux algorithmiciens…

C.3.2 Évaluation de la complexité d'algorithmes

Une évaluation rapide des complexités des algorithmes utilisés pourrait être la suivante :

- on construit la liste des tests effectués, à partir de :

```
-->Ta=[10 20 40 80] ; // les tailles des données
-->A1=[2.0056 6.6161 25.273 92.578]  // Algorithme 1
-->A2=[4.3994 10.605 24.355 54.843]  // Algorithme 2
```

- on construit des temps estimés pour différentes classes d'algorithmes : logarithmiques, linéaires, quadratiques, etc. (ici, très précisément : $O(\ln n)$, $O(n)$, $O(n \ln n)$, $O(n^2)$, et $O(n^3)$). On normalise ces temps, afin que le premier chiffre soit égal à 1 :

```
-->A1=A1 ./ A1(1)
A1 =! 1. 3.2988133 12.601217 46.159753 !
-->A2=A2 ./ A2(1)
A2 =! 1. 2.410556 5.5359822 12.466018 !
-->T1=log(Ta) ; T1=T1/T1(1)
T1 =! 1. 1.30103 1.60206 1.90309 !
-->T2=Ta ; T2=T2/T2(1)
T2 =! 1. 2. 4. 8. !
-->T3=Ta.*log(Ta) ; T3=T3/T3(1)
T3 =! 1. 2.60206 6.40824 15.22472 !
-->T4=Ta.*Ta ; T4=T4/T4(1)
T4 =! 1. 4. 16. 64. !
-->T5=Ta.*Ta.*Ta ; T5=T5/T5(1)
T5 =! 1. 8. 64. 512. !
```

- on construit les ratios, en divisant les temps mesurés par les temps estimés :

```
-->A1 ./ T1, A1 ./ T2, A1 ./ T3, A1 ./ T4, A1 ./ T5
ans =!   1.   2.5355398   7.8656334   24.25516    !
ans =!   1.   1.6494067   3.1503041    5.7699691  !
ans =!   1.   1.2677699   1.9664084    3.031895   !
ans =!   1.   0.8247033   0.7875760    0.7212461  !
ans =!   1.   0.4123517   0.1968940    0.0901558  !
-->A2 ./ T1, A2 ./ T2, A2 ./ T3, A2 ./ T4, A2 ./ T5
ans =!   1.   1.8528058   3.4555399   6.5504092  !
ans =!   1.   1.205278    1.3839955   1.5582523  !
ans =!   1.   0.9264029   0.8638850   0.8188011  !
ans =!   1.   0.602639    0.3459989   0.1947815  !
ans =!   1.   0.3013195   0.0864997   0.0243477  !
```

- dans ces résultats, les lignes dont les valeurs sont toutes voisines de l'unité indiquent la complexité effective de l'algorithme. Il s'agit de la quatrième ligne pour le premier algorithme, de la troisième pour le second, correspondant aux complexités respectives $O(n^2)$ et $O(n \ln n)$.
- il devient possible d'estimer les temps de traitement pour des objets de taille 160 : environ 370 millisecondes pour le premier algorithme, 150 millisecondes pour le second.

C.3.3 Un peu de calcul

L'exercice ne pose pas de difficultés majeures :

```
-->X=[-6.55 -7.45 -6.88 -7.12 -6.98 -7.02 -6.66 -7.34 -6.89 ...
-->-7.11 -6.95 -7.05 -6.84 -7.16 -6.99 -7.01 -6.73 -7.27] ;
-->N=count(X)
 N = 18.
-->sqrt(sum(X.^2 / N) - sum(X / N)^2)
 ans = 0.2226357
```

(Prenons simplement garde au fait que la formule nous donne σ^2, c'est-à-dire la variance, et qu'il faut extraire la racine carrée pour obtenir l'écart type.)

C.3.4 Sous-séquence maximale

Soit V le vecteur, N sa longueur. La première solution qui vient à l'esprit consiste à faire une boucle *p=1 :N* pour décrire tous les indices de débuts possibles pour les sous-séquences, puis une boucle imbriquée *q=p :N* pour décrire les indices possibles de fin de sous-séquences. Un balayage de la sous-séquence permet de calculer sa somme. Il suffit, tout au cours du processus, de conserver la meilleure valeur rencontrée pour obtenir la solution : P, Q, S, L sont les indices de début, de fin, la somme et la longueur de la séquence correspondante.

Voici une implantation de cet algorithme :

```
function Z=souseq1(V)
N=count(V)
if N==0 then
   Z=[0 0] ;
elseif N==1 then
   Z=[1 1] ;
else
   S=V(1) ;
   P=1 ;
   Q=1 ;
   L=1 ;
   for p=1 :N do
      for q=p :N do
```

```
            s=0 ;
            for l=p :q do
               s=s+V(l) ;
            end
            l = q-p+1 ;
            if s>S | s==S & l<L then
               S=s ; P=p ; Q=q ; L=l ;
            end
         end
      end
      Z=[P Q] ;
   end
```

Le test porte ici sur la somme d'une sous-séquence, et, pour deux sous-séquences de même valeur rencontrées, sur la longueur, la plus courte étant préférée.

L'algorithme donne les résultats attendus, mais ses performances sont médiocres sur des vecteurs de grande taille. L'analyse de l'algorithme montre en effet que la boucle extérieure est exécutée N fois, que la seconde boucle est exécutée de *1* à N fois, et que la boucle la plus interne porte sur des données de taille $N/2$ en moyenne. L'algorithme est donc en $O(n^3)$, et doubler la taille de son paramètre multiplie par *8* la durée d'exécution.

Il est heureusement possible d'améliorer ces performances. On peut, directement dans la boucle intérieure, calculer la somme de chaque sous-séquence, puisque celles-ci sont de longueurs croissantes, et que l'on a déjà calculé la somme des premiers termes. Le programme ainsi modifié est le suivant :

```
function Z=souseq2(V)
N=count(V)
if N==0 then
   Z=[0 0] ;
elseif N==1 then
   Z=[1 1] ;
else
   S=V(1) ;
   P=1 ;
   Q=1 ;
   L=1 ;
   for p=1 :N do
      s=0 ;
      for q=p :N do
         s=s+V(q) ;
         l = q-p+1 ;
         if s>S | s==S & l<L then
```

```
            S=s ; P=p ; Q=q ; L=l ;
          end
        end
      end
      Z=[P Q] ;
    end
```

Notre nouvelle version ne comporte plus que deux boucles imbriquées. Notre algorithme est maintenant en $O(n^2)$. Il est encore possible de faire mieux, c'est à dire $O(n)$, mais c'est un peu plus dur...

Afin de réfléchir au problème, nous allons construire une fonction qui, pour un vecteur V, nous donne la valeur, au sens décrit ci-dessus de ses sous-séquence sous la forme d'une matrice de taille $N \times N$ (la programmation de cette fonction est laissée en exercice au lecteur). Voici un petit vecteur, et la matrice correspondante :

```
-->V=[2 -7 3 9 -5 -8 2 -7 1 4 -3] ;
-->showv(V)
ans =
!  2  -5  -2   7   2  -6  -4 -11 -10  -6  -9 !
!  0  -7  -4   5   0  -8  -6 -13 -12  -8 -11 !
!  0   0   3  12   7  -1   1  -6  -5  -1  -4 !
!  0   0   0   9   4  -4  -2  -9  -8  -4  -7 !
!  0   0   0   0  -5 -13 -11 -18 -17 -13 -16 !
!  0   0   0   0   0  -8  -6 -13 -12  -8 -11 !
!  0   0   0   0   0   0   2  -5  -4   0  -3 !
!  0   0   0   0   0   0   0  -7  -6  -2  -5 !
!  0   0   0   0   0   0   0   0   1   5   2 !
!  0   0   0   0   0   0   0   0   0   4   1 !
!  0   0   0   0   0   0   0   0   0   0  -3 !
```

L'élément d'indices *(i,j)* de cette matrice nous donne la somme associée à la séquence débutant en i et se terminant en j. Que constatons-nous ?

C.4 Exercices du chapitre 4

Création de carrés magiques

Voici une solution possible du problème :

```
function M=makemagic(n)
M=zeros(n,n)
p=1
q=int((1+n)/2)
M(p,q)=1
```

```
    for k=2 :n*n do
        // nouveaux p et q
        np = 1+ modulo(p-1+n - 1, n) ;
        nq = 1+ modulo(q-1+n + 1, n) ;
        if M(np,nq) <> 0 then
            np = 1+ modulo(p-1+n + 1, n) ;
            nq = q ;
        end
        M(np,nq) = k ;
        p=np ;
        q=nq ;
    end
```

La fonction utilise k, prochaine valeur à déposer. À chaque instant, les variables p et q contiennent la position de l'élément qui vient d'être déposé. Enfin, np, nq («nouveaux» p et q) sont les positions de la prochaine valeur, calculées selon l'algorithme proposé.

C.4.1 Lecture

Quelques conseils de lecture : les fonctions dont il est question dans le chapitre, naturellement, mais aussi les pages d'aide correspondant aux mots-clefs *brackets*, *left*, *comma*, *semicolumn*, *extraction*.

C.4.2 Notation

Le résultat de notre expression (en tout cas quand la variable `i` est définie avec la valeur `%i`) est le suivant :

```
-->[1 1+i 1+2i 1+3i]
 ans =!  1.     1. + i     3.     i      4.      i !
-->size(ans)
 ans =! 1. 6. !
```

c'est-à-dire un vecteur de six éléments ! Il manque en effet, pour que l'écriture soit conforme à ce que nous espérons, les signes de multiplication :

```
-->[1 1+i 1+2*i 1+3*i]
 ans =! 1.    1. + i     1. + 2.i     1. + 3.i !
-->size(ans)
 ans =! 1. 4. !
```

C.4.3 Construction de matrice

L'une des solutions, pour obtenir le résultat demandé, est :

```
-->[eye(6,6) (1 :6)' ; 1 :7]
 ans =
! 1. 0. 0. 0. 0. 0. 1. !
! 0. 1. 0. 0. 0. 0. 2. !
! 0. 0. 1. 0. 0. 0. 3. !
! 0. 0. 0. 1. 0. 0. 4. !
! 0. 0. 0. 0. 1. 0. 5. !
! 0. 0. 0. 0. 0. 1. 6. !
! 1. 2. 3. 4. 5. 6. 7. !
```

C.4.4 Oeil gauche

Une simple boucle suffit à placer des 1 dans une matrice initialement à 0 :

```
function M=lefteye(P,Q)
M=zeros(P,Q)
for k=1 :min(P,Q) do
    M(k,Q-k+1) = 1 ;
end
```

Quelques exemples :

```
-->lefteye(5,5)
 ans =
! 0. 0. 0. 0. 1. !
! 0. 0. 0. 1. 0. !
! 0. 0. 1. 0. 0. !
! 0. 1. 0. 0. 0. !
! 1. 0. 0. 0. 0. !
-->lefteye(3,8)
 ans =
! 0. 0. 0. 0. 0. 0. 0. 1. !
! 0. 0. 0. 0. 0. 0. 1. 0. !
! 0. 0. 0. 0. 0. 1. 0. 0. !
```

C.4.5 Matrices de Hilbert

Une autre application toute simple de l'indexation :

```
function M=hilbert(n)
M=zeros(n,n)
for p=1 :n do
   for q=1 :n do
      M(p,q)=1/(p+q-1) ;
```

```
   end
end
```

Un exemple :

```
-->hilbert(5)
ans =
!  1.         0.5        0.3333333  0.25       0.2        !
!  0.5        0.3333333  0.25       0.2        0.1666667  !
!  0.3333333  0.25       0.2        0.1666667  0.1428571  !
!  0.25       0.2        0.1666667  0.1428571  0.125      !
!  0.2        0.1666667  0.1428571  0.125      0.1111111  !
```

C.4.6 Petit utilitaire

Voici la fonction de linéarisation d'un tableau.

```
function z=ravel(tab)
z=matrix(tab,1,prod(size(tab)))
```

On notera que le résultat peut parfois surprendre :

```
-->M=[1 2 3 4 5 ; 6 7 8 9 10]
 M =
!  1.   2.   3.   4.   5.!
!  6.   7.   8.   9. 10.!
-->ravel(M)
 ans =! 1. 6. 2. 7. 3. 8. 4. 9. 5. 10.!
```

C.4.7 Produit scalaire

Un exemple sur deux vecteurs :

```
-->A=[2 7 3 5]
 A =! 2. 7. 3. 5.!
-->B=[1 9 0 3]
 B =! 1. 9. 0. 3.!
-->A.*B
 ans =! 2. 63. 0. 15.!
-->sum(A.*B)
 ans = 80.
```

Une définition possible des deux fonctions :

```
-->deff('z=ps(x,y)', 'z=sum(x.*y)')
-->deff('z=angle(x,y)', 'z=acos(ps(x,y)/sqrt(ps(x,x)*ps(y,y)))')
-->ps(A,B)
 ans = 80.
-->angle(A,B)
 ans = 0.4530793
```

C.4.8 Tri ascendant

La solution la plus efficace consiste à trier l'*opposé* du vecteur V, puis à prendre l'*opposé* du résultat :

```
-->V=[ 9 2 4 10 4 8 3 1 7 6 15 7 19 5 2] ;
-->sort(V)
 ans =! 19. 15. 10. 9. 8. 7. 7. 6. 5. 4. 4. 3. 2. 2. 1. !
-->-sort(-V)
 ans =! 1. 2. 2. 3. 4. 4. 5. 6. 7. 7. 8. 9. 10. 15. 19. !
```

C.4.9 Réversion d'un vecteur

L'opération d'indexation nous apporte une solution immédiate au problème, puisqu'il suffit d'écrire `V($ :-1 :1)` pour obtenir le miroir du vecteur `V` :

```
-->V
 V =! 9. 2. 4. 10. 4. 8. 3. 1. 7. 6. 15. 7. 19. 5. 2. !
-->V($ :-1 :1)
 ans =! 2. 5. 19. 7. 15. 6. 7. 1. 3. 8. 4. 10. 4. 2. 9. !
```

C.5 Exercices du chapitre 5

C.5.1 Triangle de Pascal

Voici une solution du problème :

```
function M=Pascal(N)
M=zeros(N,N)
V=[1]
M(1,1)=V
for k=2 :N do
   V=[0 V]+[V 0] ;
   M(k,1 :k)=V ;
end
```

C.5.2 Notations

Voici un exemple de notes :

```
-->Notes=[ ...
--> 11  17  12   9  14.5  16  11.5  7.5  12.5  14    15   6   9  10.5  12
-->  8  15  11  10  13.5  15  12    5.5  11.5  14.5  13   8   7  10    11.5
--> 13  17  13  11  15    16  13    10   13    16    14  11  10  12    13 ]
```

La recherche dans la documentation sur le terme de *moyenne*, "*mean*", nous suggère la fonction `mean`. Celle-ci nous permet d'obtenir les trois informations suivantes :

```
-->mean(Notes) // moyenne générale
 ans = 12.
-->mean(Notes, 'c') // moyenne de chaque interrogation
 ans =! 11.833333 !! 11.033333 !! 13.133333 !
-->mean(Notes, 'r') // moyenne de chaque élève
 ans =
 ! 10.666667 16.333333 12. 10. 14.333333 15.666667 12.166667 7.6666667
   12.333333 14.833333 14. 8.3333333 8.6666667 10.833333 12.166667 !
```

Pour la dernière question, nous allons construire un tableau de même taille que Notes, contenant sur chaque ligne la moyenne de l'interrogation correspondante :

```
Me=mean(Notes,'c')
 Me =
! 11.833333 !
! 11.033333 !
! 13.133333 !
-->Moy=Me( :,1+0*(1 :size(Notes,2)))
 Moy =
column 1 to 7
 ! 11.833333 11.833333 11.833333 11.833333 11.833333 11.833333 11.833333 !
 ! 11.033333 11.033333 11.033333 11.033333 11.033333 11.033333 11.033333 !
 ! 13.133333 13.133333 13.133333 13.133333 13.133333 13.133333 13.133333 !
column 8 to 14
 ! 11.833333 11.833333 11.833333 11.833333 11.833333 11.833333 11.833333 !
 ! 11.033333 11.033333 11.033333 11.033333 11.033333 11.033333 11.033333 !
 ! 13.133333 13.133333 13.133333 13.133333 13.133333 13.133333 13.133333 !
column 15
 ! 11.833333 !
 ! 11.033333 !
 ! 13.133333 !
```

On peut alors comparer ces moyennes avec les notes, transformer les valeurs booléennes en nombres (par exemple, en ajoutant 0), et en déduire pour chaque élève le nombre de notes supérieures à la moyenne de la classe pour chaque interrogation :

```
-->Notes>Moy
ans =
! F T T F T T F F T T T F F F T!
! F T F F T T T F T T T F F F T!
! F T F F T T F F F T T F F F F!
-->0+(Notes>Moy)
ans =
! 0. 1. 1. 0. 1. 1. 0. 0. 1. 1. 1. 0. 0. 0. 1. !
! 0. 1. 0. 0. 1. 1. 1. 0. 1. 1. 1. 0. 0. 0. 1. !
! 0. 1. 0. 0. 1. 1. 0. 0. 0. 1. 1. 0. 0. 0. 0. !
-->sum(0+(Notes>Moy), 1)
ans =! 0. 3. 1. 0. 3. 3. 1. 0. 2. 3. 3. 0. 0. 0. 2. !
```

C.5.3 Nombres premiers

Nous allons analyser quelques unes des solutions possibles. Nous avions construit, dans un exercice antérieur, une fonction déterminant si un nombre est premier ou non. Il nous est possible d'utiliser cette fonction pour réaliser notre détermination de nombres premiers. Voici donc une première réponse :

```
function Z=premiers0(N)
Premiers=[1,2,3] ;
w=3 ; l=N-2 ;
while w<=l do
    w=w+2 ;
    if premier(w) then
        Premiers = [Premiers w] ;
    end
end
Z=Premiers ;
```

Quelle est la complexité de cette fonction ? La boucle fait varier w de *3* à *N-2*. Elle est donc en $O(n)$. La fonction *premier()*, dans l'une de ses formes données au § C.2.3, page 74, est en $O(n^{\frac{1}{2}})$. Notre fonction sera ainsi en $O(n^{\frac{3}{2}})$.

```
function Z=premiers1(k)
Premiers=[2 2 3 5 7 11 13 17 19 23 29 31 37 41] ;
w=41 ;
while w<k do
    w=w+2 ;
```

```
        if and(0<>modulo(w,Premiers)) then
            Premiers = [Premiers w] ;
        end
    end
    Z=Premiers(Premiers<=k) ;
    Z(1)=1 ;
    function [v]=premiers2(n)
    // Crible d'Eratosthène
    w=sqrt(n)
    v=1 :n
    c=2 ;
    k=2 ;
    while k<= w do
        v=v(0<>modulo(v,k)|v==k) ;
        c=c+1 ;
        k=v(c) ;
    end
    function Z=premiers3(k)
    global Premiers
    if isempty(Premiers) then
        Premiers=[2 2 3 5 7 11 13 17 19 23 29 31 37 41] ;
    end
    w=Premiers($) ;
    while w<k do
        w=w+2 ;
        if and(0<>modulo(w,Premiers)) then
            Premiers = [Premiers w] ;
        end
    end
    Z=Premiers(Premiers<=k) ;
    Z(1)=1 ;
```

C.5.4 Intégration de fonctions

La méthode utilisée dans la solution consiste à tirer un vecteur X de N éléments, dont les valeurs sont aléatoirement réparties entre A et B. On peut calculer dès lors les valeurs W de la fonction en ces divers points. On en déduit P et Q, les extremums de la fonction pour ces valeurs, et l'on peut dès lors associer au vecteur X le vecteur Y des ordonnées correspondantes, tirées au hasard entre P et Q. La comparaison des valeurs de Y et W nous donne le nombre de points du rectangle situés dans la surface comprise entre la courbe et l'axe des x. On en déduit l'intégrale demandée, qui est cette surface (obtenue par une règle de trois), à laquelle il convient

d'ajouter l'aire du rectangle limité par les droites $x = A$, $x = B$, $y = P$ et $y = 0$. Voici le texte source de la fonction :

```
function Z=MonteCarlo(F,A,B,N)
// Intégration de F entre A et B
// par la méthode de MonteCarlo
//
// 1 - Choix d'un vecteur de N elts
X=A+(B-A)*rand(1,N)
// 2 - Valeurs de la fonction
W=zeros(X) ;
for k=1 :N do
    W(k)=F(X(k)) ;
end
// 3 - Choix du rectangle
P=min(W) ; Q=max(W) ;
// 4 - ordonnées aléatoires
Y=P+(Q-P)*rand(1,N) ;
// Intégrale effective
Z=(B-A)*(Q-P)*(N-sum(1*(Y>W)))/N + (B-A)*P ;
endfunction
function r=f1(x)
r=4*x^(1/3)+3 ;
endfunction
function r=f2(x)
r=1/sqrt(x) ;
endfunction
function r=f3(x)
r=x*sqrt(1-x*x) ;
endfunction
function r=f4(x)
r=(x+6)/sqrt(x+4) ;
endfunction
```

Un calcul des trois intégrales demandées, avec 1000 points :

```
-->MonteCarlo(f1,1,8,1000)
 ans = 66.655559
-->MonteCarlo(f2,4,9,1000)
 ans = 2.0063122
-->MonteCarlo(f3,0,1,1000)
 ans = 0.3307577
-->MonteCarlo(f4,0,5,1000)
 ans = 16.686726
```

Notons encore que le lecteur curieux qui aura cherché le mot-clef "integrate" n'aura pas été déçu : il existe en effet en SCILAB une opération, `integrate()`, permettant le calcul d'une intégrale, à partir de l'expression définissant la fonction. Voici les calculs demandés, effectués au moyen de cette opération :

```
-->integrate('4*x^(1/3)+3', 'x',1,8)
 ans = 66.
-->integrate('1/sqrt(x)', 'x',4,9)
 ans = 2.
-->integrate('x*sqrt(1-x*x)', 'x',0,1)
 ans = 0.3333333
-->integrate('(x+6)/sqrt(x+4)', 'x',0,5)
 ans = 16.666667
```

C.6 Exercices du chapitre 6

C.6.1 Gags

Une expression telle que "`fun toto`" est interprétée par SCILAB comme "`fun('toto')`", c'est-à-dire l'appel de la fonction `fun` avec comme paramètre la chaîne de caractères contenant les caractères "`toto`". Dans notre expression mystère, `count` est la fonction qui compte le nombre d'éléments de son paramètre, et elle reçoit justement comme paramètre une chaîne de caractères, qui est un tableau de un élément. D'où la réponse ; aucun rapport avec un quelconque comte transsylvanien...

Cependant, même dans ce cas, la ligne est analysée en tant qu'expression SCILAB – et il se trouve qu'un point-virgule *termine* une expression. Dans le cas de "`help ;`", l'expression est interprétée comme "`help() ;`", appel de la fonction d'aide sans paramètres, qui fournit justement dans ce cas une aide sur `help`...

Deux expressions viennent à l'esprit lorsqu'il s'agit de remplacer `zeros(M)` par d'autres expressions : `M-M` et `0*M`. On peut se poser des questions sur la légitimité de leur emploi, et sur leur efficacité. Le tableau ci-dessous donne quelques éléments de réponse :

	`zeros(M)`	`M-M`	`0*M`
10 éléments	0.0941467	0.0296021	0.0231171
100 éléments	0.0984192	0.0326538	0.0258255
1000 éléments	0.1396179	0.0676727	0.0553894
10000 éléments	0.6646729	1.3598633	0.6268311
100000 éléments	30.234375	24.492188	21.425781

En pratique, à part une petite discontinuité assez inexplicable pour N=10000, on voit que nos nouvelles formulations sont dans tous les cas plus efficaces que la fonction zeros(), alors que l'on pourrait imaginer que le travail de cette dernière est plus simple... De fait, on peut constater, si l'on va chercher dans les sources du système SCILAB, que `zeros()` est en réalité une fonction définie, qui calcule son résultat comme... `0*ones(M)` !

C.6.2 L'indice erroné

Les deux expressions `[A,B]` et `[A ;B]` fournissent des tableaux différents. Le premier est de dimensions *(1,12)*, le second *(2,6)*. Les éléments de `A` et `B` n'y sont pas rangés dans le même ordre – d'où les différences entre les résultats.

C.6.3 Meilleure expression

Les expressions proposées correspondent en fait aux combinaisons deux à deux des deux approches suivantes :

- vecteur résultat initialement vide, ou encore vecteur de taille initiale 10000 ;
- boucle de remplissage de 1 à 10000, ou encore de 10000 à 1.

Elles nous fournissent les "timings" suivants :

```
-->chrono("V=[] ;for k=1 :10000 do V(k)=2*k-1 ; end ;")
2038.75 ms.
-->chrono("V=zeros(1,10000) ;for k=1 :10000 do V(k)=2*k-1 ; end ;")
266.71875 ms.
-->chrono("V=[] ;for k=10000 :-1 :1 do V(k)=2*k-1 ; end ;")
266.40625 ms.
-->chrono("V=zeros(1,10000) ;for k=10000 :-1 :1 do V(k)=2*k-1 ; end ;")
266.875 ms.
```

Il semble paradoxal qu'une telle disparité puisse se produire entre les durées des quatre expressions. L'inefficacité est ici liée au mode de remplissage du tableau. Dans les expressions 2 et 4, le tableau préexiste au moment du remplissage. Dans l'expression 3, le tableau est initialement vide, mais la première affectation a pour effet de créer l'élément de numéro 10000, donc le vecteur dans son intégralité. Dans l'expression 1 au contraire, le vecteur initialement vide est transformé en un vecteur de 1 élément, puis de 2 éléments, puis de 3, etc. Toutes ces opérations nécessite de recopier 1, puis 2, puis trois, etc, jusqu'à 9999 éléments. Du fait de cette copie, l'algorithme effectif de la boucle n'est plus en $O(n)$, mais en $O(n^2)$, et cette différence est sensible car le nombre d'éléments n'est pas petit.

La conclusion est qu'il vaut mieux créer en une seule fois les tableaux, lorsqu'il faut les remplir élément par élément...

Le second problème, au contraire, est un de ces excellents exemples où *l'on ne peut rien dire a priori*. Il s'agit uniquement d'une question de performance de l'implantation, et une expression plus lente qu'une autre dans la version 2.5 du logiciel peut devenir plus efficace dans la version 2.6. En l'occurrence, les tests nous disent, pour un vecteur de 10000 éléments dont en gros 5000 sont positifs :

```
-->V=rand(1,10000)-0.5 ;
-->sum(max(V,0))
ans = 1221.9478
-->sum(V(V>0))
ans = 1221.9478
```

```
-->chrono("sum(max(V,0))") ;
1.4855957 ms.
-->chrono("sum(V(V>0))")
1.9519043 ms.
```

C.6.4 Diagonale

Voici manifestement ce qu'il convient d'appeler un exercice de créativité. Quelques solutions :

```
-->diag(diag(M))
-->M.*eye(M)
-->tril(triu(M))
```

C.6.5 Tours de Hanoi

C.6.5.1 Première partie

Nous allons essayer de montrer comment transformer un programme existant, en l'adaptant à de nouvelles contraintes, tout en respectant (plus ou moins) les règles énoncées au long du chapitre. On peut réécrire ainsi notre fonction `Hanoi()` :

```
function Hanoi(N)
Hanoi_Conf(N,[1 :N],[],[])
Hanoi_R(N,1,3)
endfunction
function Hanoi_R(N,A,B)
if N>0 then
    C=6-A-B ;
    Hanoi_R(N-1,A,C) ;
    Hanoi_MV(A,B) ;
    Hanoi_R(N-1,C,B) ;
end
endfunction
function Hanoi_Conf(N,P1,P2,P3)
endfunction
function Hanoi_MV(A,B)
disp(string(A)+" -> "+string(B)) ;
endfunction
```

Sous cette nouvelle écriture, notre application reste fonctionnellement identique à la version fournie dans le cours. Nous avons simplement décomposé le programme en :
- un programme principal, `Hanoi()`, qui effectue une initialisation du système, et fait appel à la procédure de calcul, `Hanoi_R()` ;
- une procédure de calcul, `Hanoi_R()`, qui fait elle-même appel à la procédure d'impression de résultats ;

– une procédure d'initialisation, `Hanoi_Conf()`, qui ne fait pas grand chose pour l'instant ;
– une procédure d'impression, `Hanoi_MV()`, qui fonctionne à l'identique.

Nous allons maintenant choisir une structure de données pour représenter la configuration des tours. Il y a trois tours, chacune pouvant supporter jusqu'à N disques. Une matrice de taille *(3,N)* permet de représenter n'importe quelle configuration, en convenant de désigner les disques par leur numéro (de 1, le plus grand des disques, jusqu'à N, le plus petit), et les emplacements libres par 0. Dans une ligne de la matrice, le premier élément désignera le disque qui est au sommet, le second celui qui est en dessous, etc. Pour une configuration de 5 disques, initialement empilés dans l'ordre sur la tour numéro 1, la représentation est la suivante :

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

Lorsque le disque du sommet de la tour 1 a été transféré sur la tour 3, la configuration devient :

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

Voici les nouvelles versions des procédures *d'initialisation* et de *transfert*, qui créent et gèrent cette représentation des données. Pour cette application, nous avons choisi de déclarer *globale* la matrice qui représente la configuration courante des tours.

```
function Hanoi_Conf(N,P1,P2,P3)
global Hanoi_data
Hanoi_data=zeros(3,N) ;
Hanoi_data(1,count(P1) :-1 :1)=P1
Hanoi_data(2,count(P2) :-1 :1)=P2
Hanoi_data(3,count(P3) :-1 :1)=P3
endfunction
function Hanoi_MV(A,B)
// un disque passe de A en B
global Hanoi_data
D=Hanoi_data(A,1) ; // n° du disque
Hanoi_data(A, :)=[Hanoi_data(A,2 :$),0] ;
Hanoi_data(B, :)=[D,Hanoi_data(B,1 :$-1)] ;
disp("Le disque "+string(D)+" passe de "+ ...
        string(A)+" -> "+string(B)) ;
endfunction
```

La représentation des données est assez facile à générer. Le numéro du disque à transférer est immédiatement disponible. Enfin, transférer un disque ne demande pas de manipulations très complexes dans la représentation. Voici un exemple :

```
-->Hanoi(3,1,3)
Le disque 3 passe de 1 -> 3
Le disque 2 passe de 1 -> 2
Le disque 3 passe de 3 -> 2
Le disque 1 passe de 1 -> 3
Le disque 3 passe de 2 -> 1
Le disque 2 passe de 2 -> 3
Le disque 3 passe de 1 -> 3
```

L'impression de la configuration finale nous donne :

```
-->Hanoi_data
Hanoi_data =
! 0. 0. 0.!
! 0. 0. 0.!
! 3. 2. 1.!
```

C.6.5.2 Seconde partie

Le problème semble moins simple. Pourtant, une première réflexion nous convainc que le problème est réellement de déplacer la plus grande des tours de la position de départ vers la position d'arrivée. Pour ce faire, il convient de transférer tout ce qui est "au dessus" vers la pile où ne doit pas aller le disque. Pour réaliser cette opération, on peut répéter l'algorithme, en ignorant le premier disque, et ainsi de suite.

Comment représenter une position ? On peut naturellement utiliser la représentation choisie dans la première partie. On peut aussi en adopter une autre, plus appropriée au problème. Dans celle-ci, un vecteur de N éléments représente les N disques, et la valeur de chaque entrée indique la tour (1, 2 ou 3) sur laquelle se trouve le disque correspondant. Ainsi, 5 disques empilés en position 1 sont représentés par le vecteur `[1 1 1 1 1]`, et le transfert du disque du dessus en position 3 change ce vecteur en `[1 1 1 1 3]`.

Notre nouvelle programmation reflète cette structure de données :

```
function Hanoi(N)
Hanoi_Conf(ones(1,N))
Hanoi_R(N,1,3)
endfunction
function Hanoi_Conf(Conf)
global Hanoi_C Hanoi_N
Hanoi_C=Conf ;
Hanoi_N=count(Conf) ;
endfunction
function Hanoi_MV(A,B)
// un disque passe de A en B
global Hanoi_C Hanoi_N
```

```
// Le disque a déplacer est le dernier de la pile A
for k=1 :Hanoi_N do
   if Hanoi_C(k)==A then D=k ; end
end ;
Hanoi_C(D)=B ;
disp("Le disque "+string(D)+" passe de "+ ...
        string(A)+" -> "+string(B)) ;
endfunction
```

Cette nouvelle version de l'application fonctionne identiquement à la précédente, les seuls changements concernant la structure de données utilisée pour représenter la configuration des tours.

La fonction suivante, qui permet d'atteindre une position spécifiques des disques, reflète l'algorithme exposé ci-dessus :

```
function HanGoal(C,S)
// Parvenir à la configuration C
// à partir du disque S.
global Hanoi_C Hanoi_N
for k=S :Hanoi_N do
    if C(k) <> Hanoi_C(k) then
        // le disque k n'est pas bien placé
        // On va libérer tous ceux qui sont au dessus
        P=6-C(k)-Hanoi_C(k) ; // P est le piquet libre
        New=C ;
        for w=k+1 :Hanoi_N do New(w)=P ; end ;
        HanGoal(New,k+1) ;
        // déplacer le disque k
        Hanoi_MV(Hanoi_C(k),C(k)) ;
    end ;
end ;
endfunction
```

Voici enfin un exemple de son utilisation :

```
-->Hanoi_Conf([1 3 2 3])
-->HanGoal([1 2 2 1],1)
Le disque 3 passe de 2 -> 1
Le disque 4 passe de 3 -> 1
Le disque 2 passe de 3 -> 2
Le disque 4 passe de 1 -> 3
Le disque 3 passe de 1 -> 2
Le disque 4 passe de 3 -> 1
```

On part ici d'une configuration dans laquelle le disque 1 est en 1, le disque 2 en 3, le disque 3 en 2 et le disque 4 en3 ; on veut obtenir une position dans laquelle le disque 1 est en 1, le disque 2 en 2, le disque 3 en 2, et le disque 4 en 1. L'exécution de la procédure indique les déplacements à accomplir.

C.7 Exercices du chapitre 7

C.7.1 Tableau de fils

Voici un exemple, choisi aussi simple que possible : on définit 100 valeurs entre $\pi/4$ et $9\pi/4$, qui vont fournir les coordonnées des points sur le cercle, et 100 valeurs comprises entre 0 et 1, qui fourniront 100 points de la droite. La fonction `xsegs()` permet de tracer en un seul appel l'ensemble des segments :

```
function gf03()
xbasc()
// Points sur un cercle
v=(%pi/4)+linspace(0,2*%pi,100)
S=sin(v) ;
C=cos(v) ;
m1=0.5*S m2=0.5*C
// Points sur une droite
w=linspace(0,1,100)
xsegs([m1 ;w],[m2 ;w])
```

C.7.2 Palettes graphiques

On peut obtenir un dégradé entre deux couleurs par une simple interpolation linéaire. Cependant, les couleurs intermédiaires semblent un peu plus sombres que les couleurs extrêmes. Une solution, qui donne des résultats relativement agréables à l'oeil, consiste à prendre les *racines carrées* des teintes ainsi obtenues. Voici, à titre d'exemple, le calcul d'une palette de 129 couleurs, dont C1 et C2 (deux vecteurs de trois valeurs comprises entre 0 et 1) définissent les teintes extrêmes :

```
C1= [1 0 0] ; // Rouge
C2= [1 1 0] ; // Jaune
K=129 // Nb. de couleurs
cmap=zeros(K,3) ;
for k=1 :K do
   cmap(k, :)=sqrt(C2*((k-1)/(K-1))+C1*((K-k)/(K-1)))
end ;
xset("colormap",cmap) ;
```

C.7.3 Triangle de Serpinsky

Une solution :

```
function Serp(N)
// Courbe de Serpinsky
//
if N<1 | N>30 then
    error("Paramètre incorrect") ;
end
// Initialisation du graphique
if ~ isdef("Win") then Win=0 ; end
xset("window",Win) ;
driver("X11") ;
xbasc() ;
square(-1,-1,1,1)
Black=0 ; xset("pattern",Black) ; xrect(-1,1,2,2)
// Choix des points
a=%pi/2
b=%pi*7/6
c=%pi*11/6
A=[cos(a) sin(a)] ;
B=[cos(b) sin(b)] ;
C=[cos(c) sin(c)] ;
minim=sin(b) ; maxim=1-minim ;
// Choix de la palette
if ~ isdef("Colors") then
    Colors=[1 0 0    // Première couleur
            1 1 0 ] ; // Seconde couleur
end ;
C1=Colors(1, :) ;
C2=Colors(2, :) ;
K=129 // Nb. de couleurs
cmap=zeros(K,3) ;
for k=1 :K do
    cmap(k, :)=sqrt(C2*((k-1)/(K-1))+C1*((K-k)/(K-1)))
end ;
xset("colormap",cmap) ;
// Tracé des courbes
serp(A,B,C,1) ;
endfunction
```

```
function serp(A,B,C,n)
//
// Tracé
if (n>=N) then return ; end ;
A1=(B+C)/2 ;
B1=(C+A)/2 ;
C1=(A+B)/2 ;
triangle(A1,B1,C1) ;
n=n+1 ;
serp(A,B1,C1,n) ;
serp(B,C1,A1,n) ;
serp(C,A1,B1,n) ;
endfunction
function triangle(A,B,C)
// Tracé d'un triangle
x=[A(1),B(1),C(1)] ;
y=[A(2),B(2),C(2)] ;
// On choisit pour la couleur une
// valeur liée à la position en "x"
k=max(x) ;
xset("pattern",1+(K-1)*(k-minim)/maxim) ;
xpoly(x,y,"lines",1) ;
endfunction
```

C.7.4 Courbe du Dragon

Une solution :

```
function Dragon(N)
// Courbe du Dragon
//
if N<1 | N>30 then
    error("Paramètre incorrect") ;
end
// Initialisation du graphique
if ~ isdef("Win") then Win=0 ; end
xset("window",Win) ;
// driver("X11") ;
xbasc() ;
square(0,0,1,1)
Black=0 ; xset("pattern",Black) ; xrect(0,1,1,1) ;
```

```
// Choix des points
A=[0.3 0.4] ;
B=[0.7 0.8] ;
// Choix de la palette
if ~ isdef("Colors") then
    Colors=[1 0 0    // Première couleur
            1 1 0 ] ; // Seconde couleur
end ;
C1=Colors(1, :) ;
C2=Colors(2, :) ;
K=129 // Nb. de couleurs
cmap=zeros(K,3) ;
for k=1 :K do
    cmap(k, :)=sqrt(C2*((k-1)/(K-1))+C1*((K-k)/(K-1)))
end ;
xset("colormap",cmap) ;
// Tracé des courbes
dragon(N,A,B,0,1) ;
endfunction
function dragon(N,S,D,u,v)
//
// Tracé
N=N-1 ;
c=(u+v)/2 ;
if N<=0 then
    xset("pattern",round(1+c*(K-1))) ;
    xpoly([S(1),D(1)], [S(2),D(2)], "lines", 0)
else
    I=[((S(1)+D(1))/2 + (D(2)-S(2))/2) ...
       ((S(2)+D(2))/2 - (D(1)-S(1))/2)]
    dragon(N,S,I,u,c) ;
    dragon(N,D,I,v,c) ;
end ;
endfunction
```

C.7.5 La tortue Logo

C.8 Exercices du chapitre 8

C.8.1 Chiffres romains

Voici les deux fonctions de conversion :

```
function Z=rom2dec(N)
Z=0;
N=" "+strsubst(N, " ", "")
while length(N)>1 do
    if strindex(N, " M") then
        Z=Z+1000; N=strsubst(N, " M", " ");
    elseif strindex(N, " CM") then
        Z=Z+900; N=strsubst(N, " CM", " ");
    elseif strindex(N, " D") then
        Z=Z+500; N=strsubst(N, " D", " ");
    elseif strindex(N, " CD") then
        Z=Z+400; N=strsubst(N, " CD", " ");
    elseif strindex(N, " C") then
        Z=Z+100; N=strsubst(N, " C", " ");
    elseif strindex(N, " XC") then
        Z=Z+90; N=strsubst(N, " XC", " ");
    elseif strindex(N, " L") then
        Z=Z+50; N=strsubst(N, " L", " ");
    elseif strindex(N, " XL") then
        Z=Z+40; N=strsubst(N, " XL", " ");
    elseif strindex(N, " X") then
        Z=Z+10; N=strsubst(N, " X", " ");
    elseif strindex(N, " IX") then
        Z=Z+9; N=strsubst(N, " I", " ");
    elseif strindex(N, " V") then
        Z=Z+5; N=strsubst(N, " V", " ");
    elseif strindex(N, " IV") then
        Z=Z+4; N=strsubst(N, " IV", " ");
    elseif strindex(N, " I") then
        Z=Z+1; N=strsubst(N, " I", " ");
    else
        error("Caractère illégal :"+N);
    end;
end;
function Z=dec2rom(N)
if N<0 | N > 10000 then
    error("Valeur hors limites")
end
Z="";
while N > 0 do
```

```
    if N >= 1000 then
        N=N-1000 ; Z=Z+"M" ;
    elseif N >= 900 then
        N=N-900 ; Z=Z+"CM" ;
    elseif N >= 500 then
        N=N-500 ; Z=Z+"D" ;
    elseif N >= 400 then
        N=N-400 ; Z=Z+"CD" ;
    elseif N >= 100 then
        N=N-100 ; Z=Z+"C" ;
    elseif N >= 90 then
        N=N-90 ; Z=Z+"XC" ;
    elseif N >= 50 then
        N=N-50 ; Z=Z+"L" ;
    elseif N >= 40 then
        N=N-40 ; Z=Z+"XL" ;
    elseif N >= 10 then
        N=N-10 ; Z=Z+"X" ;
    elseif N >= 9 then
        N=N-9 ; Z=Z+"IX" ;
    elseif N >= 5 then
        N=N-5 ; Z=Z+"V" ;
    elseif N >= 4 then
        N=N-4 ; Z=Z+"IV" ;
    elseif N >= 1 then
        N=N-1 ; Z=Z+"I" ;
    end ;
end ;
```

C.8.2 Palindromes

Voici une solution possible :

```
function [R,C]=palindrome(S)
// Ésope reste ici et se repose.
// Équivalence des lettres
stra="àÀâÂäÄ"
strc="çÇ"
stre="éèêÉÈÊëË"
stri="îÎïÏ"
stro="ôÔöÖ"
```

```
stru="ùÙûÛüÜ"
// Table de conversion
alf=ravel(str2code("abcdefghijklmnopqrstunwxyz"))
Tab=zeros(1,512)
Tab(alf)=alf
Tab(str2code(stra))=str2code("a")
Tab(str2code(strc))=str2code("c")
Tab(str2code(stre))=str2code("e")
Tab(str2code(stri))=str2code("i")
Tab(str2code(stro))=str2code("o")
Tab(str2code(stru))=str2code("u")
// abs() passe les majuscules en minuscules
v=Tab(abs(str2code(S)))
v=v(v<>0) ;
C=code2str(v)
R=%t ;
n=count(v) ;
for k=1 :int(n/2) do
    if v(k)<>v(n-k+1) then R=%f ; end ;
end ;
```

Voici une utilisation de la fonction :

```
-->palindrome("Ésope reste ici et se repose")
 ans = T
-->[r,s]=palindrome("Ésope reste ici et se repose")
 s = esoperesteicietserepose
 r = T
-->[r,s]=palindrome("Ésope reste ici et roupille")
 s = esoperesteicietroupille
 r = F
```

C.9 Exercices du chapitre 9

C.9.1 Tableaux creux

Nous utilisons ici le fait que le type d'une tableau creux est 5 (celui d'un tableau plein est 4). Voici les opérations demandées :

```
function Z=issparse(A)
Z=type(A)==5 ;
endfunction
```

La fonction `spcount()` va de même tester le type de son paramètres, et appliquer deux algorithmes différents en fonctions de celui-ci :

```
function Z=spcount(A)
if type(A)==5 then
    Z=sum(spones(A)) ;
else
    Z=sum(A<>0) ;
end
```

De même, la fonction `spdiag()` teste les dimensions de son opérande, et essaye de produire un résultat de manière économique :

```
function Z=spdiag(A)
D=size(A)
if D(1)==1 | D(2)==1 then
    N=prod(D) ;
    Z=spzeros(N,N) ;
    for k=1 :N do
        Z(k,k)=A(k) ;
    end
else
    N=min(D)
    Z=spzeros(N,1) ;
    for k=1 :N do
        Z(k,1)=A(k,k) ;
    end
end
endfunction
```

C.10 Exercices du chapitre 10

C.11 Tours de Hanoi

Voici la version complète du programme, réalisant l'animation graphique des tours de Hanoi. Le fichier `han.sci`, comporte l'ensemble des fonctions nécessaires.

```
function han(W)
  // Tours de Hanoi
  // J.J.Girardot, Août 2000
  //
  global Hcnt
  Mx=16 // Nb. max de tours
```

```
// Vérifications
C=abs(W) ;
if C>Mx then error("Valeur incorrecte") ; end ;
if C==0 then C=Mx ; end
//
// Choix des couleurs
// Dégradé de rouge à jaune
// Blue = 0
cmap=zeros(Mx,3) ;
// Red = 1
cmap( :,1)=1 ;
// Green : de 0 à 1
cmap( :,2)=matrix(min(1,(0 :Mx-1)/max(1,C-1)),Mx,1)
// On ajoute blanc et noir
White=size(cmap,1)+1 ;
Black=White+1 ;
cmap=[cmap ; 1 1 1 ; 0 0 0] ;
// IMPORTANT : sélection d'un driver
// qui n'enregistre pas...
driver("X11") ;
// Nettoyage
xbasc() ;
// Choix des couleurs
xset("colormap",cmap)
// Dimensions Dim=1000 ;
// On va opérer dans un carré
square(0,0,Dim,Dim) ;
// Définition de quelques valeurs par défaut
if ~ isdef("Delai") then Delai=0 ; end
if ~ isdef("LongDelai") then LongDelai=0 ; end
if ~ isdef("Trace") then Trace=0 ; end
Trace=round(Trace) ;
// Tracer les progrès ?
TracePr = modulo(Trace,2) == 1 ;
// Tracer les numéros de disques ?
TraceDk= modulo(int(Trace/2),2) == 1 ;
delta=2 ; // pas des déplacements pour l'animation
// Positions verticales des disques
Dp=10*(10+4*(1 :Mx))
// Couleurs des disques
```

```
Dc=1 :C
// Ajustement des largeurs des disques
Dl=round(linspace(140,20,max(2,C)))
// Ajustement de la hauteur des tiges
Dh=30 ; // hauteur d'un disque
Ba=100 ; // position du support
Cl=4 ; // nettoyage
Pp=[200, 500, 800] ; // Position des tiges
Pw=10 ; // largeur des tiges
Ph=40*C+60 ; // hauteur d'une tige
Pe=Ph+Ba ; // sommet d'une tige
Dtop=Pe+80 ; // position haute d'un disque
// Avancement
QPx=10 ; QPy=Dim-QPx ; QPh=20 ; QPl=Dim-2*QPx ;
//
hanboard() ; // dessin du support
// Dessin initial des disques
for k=1 :C do
    xpause(LongDelai) ;
    hanmvdown(1,k,k) ;
end ; //
xpause(2*LongDelai) ;
//
Hmax=2^C-1 ; // Nb total de déplacements
//
if (W > 0) then
    // Appel de type han(4)
    Hcnt=Hmax ;
    hanoi(1 :C,[],[],C,1,3) ;
elseif (W == 0) then
    // Mode "show"
    hanshow() ;
else
    // Mode "demo", par exemple han(-4)
    while %t do
        Hcnt=Hmax ;
        hanoi(1 :C,[],[],C,1,3) ;
        Hcnt=Hmax ;
        hanoi([],[],1 :C,C,3,2) ;
        Hcnt=Hmax ;
```

```
          hanoi([],1 :C,[],C,2,1) ;
          Hcnt=Hmax ;
          hanoi(1 :C,[],[],C,1,2) ;
          Hcnt=Hmax ;
          hanoi([],1 :C,[],C,2,3) ;
          Hcnt=Hmax ;
          hanoi([],[],1 :C,C,3,1) ;
      end
  end
//
endfunction
function hanmove(D,A,h1,B,h2)
  hanmvup(A,D,h1) ;
  hanmvh(A,B,D) ;
  hanmvdown(B,D,h2) ;
endfunction
function hanoi(D1,D2,D3,N,A,B)
  // Hanoi - principale
  // Faire passer N disques de A vers B
  if N == 1 then
      // un disque a déplacer de A vers B
      [D,h1,h2,D1,D2,D3]=hanmvlast(D1,D2,D3,A,B)
      hanmove(D,A,h1,B,h2)
      if TracePr then
          hanprogres() ;
      end ;
      xpause(LongDelai) ;
  else
      C=6-A-B ;
      // N-1 disques de A vers C
      hanoi(D1,D2,D3,N-1,A,C) ;
      [D1,D2,D3]=hanmvdsk(D1,D2,D3,N-1,A,C) ;
      // 1 disque de A vers B
      hanoi(D1,D2,D3,1,A,B) ;
      [D1,D2,D3]=hanmvdsk(D1,D2,D3,1,A,B) ;
      // N-1 disques de C vers B
      hanoi(D1,D2,D3,N-1,C,B) ;
      [D1,D2,D3]=hanmvdsk(D1,D2,D3,N-1,C,B) ;
  end
endfunction
```

```
function hanboard()
  // Tracé du Cadre
  xset("pattern",Black) ;
  xfrect(0,Dim,Dim,Dim) ;
  w=2 ;
  xset("pattern",White) ;
  xfrect(0+w,Dim-w,Dim-2*w,Dim-2*w) ;
  xset("pattern",Black) ;
  Ka=int(Ba/2) ;
  xfrect(Ka,Ba,Dim-Ba,Ka) ;
  xfrect(Pp(1),Pe,Pw,Ph) ;
  xfrect(Pp(2),Pe,Pw,Ph) ;
  xfrect(Pp(3),Pe,Pw,Ph) ;
endfunction
function hanprogres()
  global Hcnt
  xset("pattern",Black) ;
  xfrect(QPx,QPy,QPl,QPh) ;
  Hcnt=Hcnt-1 ;
  xset("pattern",1) ;
  xfrect(QPx,QPy,QPl*(1-Hcnt/Hmax),QPh) ;
endfunction ;
function hanmvup(nplot,num,hdep)
  // Point de départ
  w=Dl(num) ;
  Co=Dc(num) ;
  px=Pp(nplot) ;
  x=px-w y=Dp(hdep) ;
  Y=Dtop ;
  Dw=w+w+Pw ;
  while y<Y do
      y=y+delta ;
      xset("pattern",White) ;
      xfrect(x,y-Dh,Dw,delta) ;
      if y<Pe+Dh then
          xset("pattern",Black) ;
          xfrect(px,y-Dh,Pw,delta) ;
      end
      xset("pattern",Co) ;
      xfrect(x,y,Dw,delta) ;
```

```
      xpause(Delai) ;
    end
  endfunction
  function hanmvdown(nplot,num,hdep)
    // Point de départ
    w=Dl(num) ;
    Co=Dc(num) ;
    px=Pp(nplot) ;
    x=px-w ;
    Y=Dp(hdep) ;
    y=Dtop ;
    Dw=w+w+Pw ;
    while y>Y do
        xset("pattern",White) ;
        xfrect(x,y,Dw,delta) ;
        if y<Pe then
            xset("pattern",Black) ;
            xfrect(px,y,Pw,delta) ;
        end
        xset("pattern",Co) ;
        xfrect(x,y-Dh,Dw,delta) ;
        y=y-delta ;
        xpause(Delai) ;
    end
    // Nettoyage final des
    // petits résidus éventuels
    xset("pattern",White) ;
    xfrect(0+Cl,Dtop+Cl,Dim-2*Cl,Dh+2*Cl) ;
    // Numéro du disque ?
    if TraceDk then
        xset("pattern",Black) ;
        if num>9 then dd=6 ; else dd=0 ; end ;
        xstring(px-dd,y-30,string(num)) ;
    end
  endfunction
  function hanmvh(splot,dplot,num)
  // Déplacement horizontal des disques
    // Point de départ
    w=Dl(num) ;
    Co=Dc(num) ;
```

```
    xs=Pp(splot)-w ;
    xd=Pp(dplot)-w ;
    y=Dtop ;
    Dw=w+w+Pw ;
    if splot < dplot then
        // déplacement a droite
        while xs<xd do
            xset("pattern",White) ;
            xfrect(xs,y,delta,Dh) ;
            xset("pattern",Co) ;
            xfrect(xs+Dw,y,delta,Dh) ;
            xs=xs+delta ;
            xpause(Delai) ;
        end
    else
        // déplacement a gauche
        while xs>=xd do
            xset("pattern",White) ;
            xfrect(xs+Dw,y,delta,Dh) ;
            xset("pattern",Co) ;
            xfrect(xs,y,delta,Dh) ;
            xs=xs-delta ;
            xpause(Delai) ;
        end ;
    end
  endfunction
  function [D1,D2,D3]=hanmvdsk(D1,D2,D3,N,A,B)
  // N disques sont déplacés de A vers B
  // Mise à jour des tableaux D1, D2, D3
    // d'où viennent les disques ?
    if A == 1 then
        V=D1($-N+1 :$) ; D1($-N+1 :$)=[] ;
    elseif A == 2 then
        V=D2($-N+1 :$) ; D2($-N+1 :$)=[] ;
    else
        V=D3($-N+1 :$) ; D3($-N+1 :$)=[] ;
    end ;
    // où vont-ils ?
    if B == 1 then
        D1 = [D1 V] ;
```

```
    elseif B == 2 then
        D2 = [D2 V] ;
    else
        D3 = [D3 V] ;
    end ;
  endfunction
  function [D,hs,hd,D1,D2,D3]=hanmvlast(D1,D2,D3,A,B)
  // Gestion des listes de disques.
  // Un disque va de A vers B
  // mise à jour des tableaux D1, D2, D3
  //
    // d'où vient le disque ?
    if A == 1 then
        D=D1($) ; hs=size(D1,2) ; D1(hs)=[] ;
    elseif A == 2 then
        D=D2($) ; hs=size(D2,2) ; D2(hs)=[] ;
    else
        D=D3($) ; hs=size(D3,2) ; D3(hs)=[] ;
    end ;
    // où va le disque ?
    if B == 1 then
        D1=[D1 D] ; hd=size(D1,2) ;
    elseif B == 2 then
        D2=[D2 D] ; hd=size(D2,2) ;
    else
        D3=[D3 D] ; hd=size(D3,2) ;
    end
  endfunction
  function hanshow()
    // La position de départ est affichée
    A=1 ; B=2 ; C=3 ;
    disp("Type ""Return""") ;
    readc_() ;
    // Position intermédiaire
    hanboard() ;
    // Une configuration choisie au hasard
    Pf=[C,B,B,B,A,C,A,B,B,A,C,B,B,B,B,B] ;
    Hf=[1,1,2,3,1,2,2,4,5,3,3,6,7,8,9,10] ;
    for k=1 :16 do
        hanmove(k,A,16,Pf(k),Hf(k)) ;
```

```
    end
    hanmvup(C,11,3) ;
    disp("Type ""Return""") ;
    readc_() ;
    // Position finale
    hanboard() ;
    for k=1 :16 do
        hanmove(k,A,01,C,k) ;
    end
  endfunction
```

Fin du listing.

Table des figures

1.1 Fonction $y = x^2$ . . . 12
1.2 Gestion des fenêtres graphiques . . . 15
1.3 Géométrie des fenêtres graphiques . . . 15
1.4 Tours de Hanoi . . . 16
1.5 Fonction $z = \sin x + \sin y$ . . . 19
1.6 Tableau de fils . . . 20
1.7 Triangle de Serpinsky d'ordre 8 . . . 21
1.8 Flocon de Koch, pour N=0,1,2 et 3. . . . 22
1.9 Courbe du Dragon . . . 23

2.1 Table des codes ASCII . . . 29

3.1 Performances des opérations sur listes . . . 42
3.2 Opérations de création de tableaux creux . . . 44

4.1 Types internes de données . . . 48

Bibliographie

[1] Gérald Grandpierre and Richard Cotté. *Mathématiques et Graphismes.* Éditions du P.S.I, B.P. 86, 77402 Lagny, 1985.

Index

" (caractère), 25
' (caractère), 25
+ (concaténation), 26
+ (opération), 28
écran graphique, 13

add (fonction), 20
APL (langage), 39
apostrophe, 25
ASCII (code), 28
ascii (opération), 28

blanc (couleur), 17
bleu (couleur), 17

C, 51
call (opération), 52
casse, 32
chaîne de caractères, 25
clear (opération), 58
concaténation, 26, 28
convstr (opération), 32
creuse (matrice), 43

disp (opération), 33
double-apostrophe, 25
double-quote, 25
Dragon (courbe du), 24
driver (grahique), 16
driver (opération), 16

emptystr (opération), 26
Euclide, 76

fenêtre
  d'interaction, 14
  graphique, 11, 14
Fibonacci, 51
Fortran, 51
full (opération), 44

Gif, 16
Girardot (Jean-Jacques), 1
glyphe, 27

Hanoi (Tours), 16

implantation (de Scilab), 47
isdef (opération), 57
isoview (opération), 15
issparse (opération), 45, 102

jaune (couleur), 17

Koch (flocon de), 22, 24

length (opération), 26, 40
link (opération), 52
list (opération), 39
listcat (opération), 42
liste, 39
Logo (tortue), 24

matrice creuse, 43

noir (couleur), 17
null (opération), 41, 58

ones (opération), 44
orange (couleur), 17

Pérec (George), 36
palette graphique, 17
palindrome, 36
part (opération), 30
PGCD, 76
pixel, 13, 16
plot (opération), 11
plot3d (opération), 19
plot3d1 (opération), 20

poly (opération), 48
polynôme, 48
Postscript, 16
programme
    objet, 52
    source, 52

quote, 25

résolution, 13
readc_ (opération), 32
rouge (couleur), 17

Serpinsky (triangle de), 21
size (opération), 40
sparse matrix, 43
spcount (opération), 45, 103
spdiag (opération), 45, 103
speye (opération), 44
spones (opération), 44
sprand (opération), 44
spzeros (opération), 44
square (opération), 14, 15
str2cod (opération), 27
strcat (opération), 30
strindex (opération), 30
string (opération), 26
stripblanks (opération), 31
strsubst (opération), 31

tableau
    vide, 58
tableau généralisé, 39
tortue Logo, 24
type (opération), 47, 77
types de données, 47

valeur nulle, 41
vert (couleur), 17
vide (tableau), 58

X11, 16
xbasc (opération), 14, 15
xdel (opération), 15
Xfig, 16
xfrect (opération), 16, 17
xpause (opération), 18
xsegs (opération), 14
xselect (opération), 15
xset (opération), 15, 17
xsetech (opération), 15