Une introduction à la programmation par SCILAB

Première Partie

ÉLÉMENTS DE BASE

Jean-Jacques Girardot

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

Mots-clefs : introduction à la programmation, SCILAB
Première partie. Date de création : 27 septembre 2000.

Table des matières

Avant-propos

0.1 Introduction

Ce document est une brève introduction à la programmation. Pour des raisons qui sont détaillées plus loin, cette introduction s'effectue au travers du logiciel SCILAB. Le langage de programmation particulier utilisé comme support de la pensée dans un cours d'informatique ne devrait jouer qu'un rôle assez secondaire vis à vis des concepts enseignés. Ce n'est malheureusement presque jamais le cas, et, en particulier dans le cas des "initiations" à l'informatique, ce choix du langage joue un rôle prépondérant. L'expérience montre que l'apprenant est en effet rapidement préoccupé (voire envahi) par les seuls aspects de mise en œuvre du langage choisi, et que les concepts, parfois de haut vol, que l'enseignant vise à mettre à sa disposition sont largement ignorés (lorsqu'ils ne sont pas traités par le mépris).

Nous voulons construire des programmes solides, et le langage de programmation va nous permettre de créer nos briques. Mais il ne sert à rien de construire sur du sable ; la compréhension de la nature et de la structure des données, la connaissance de l'algorithmique constituent les fondations de toute construction correcte. C'est un passage difficile, mais obligé.

En bref, on ne s'improvise pas programmeur, et ceux qui prétendent le contraire n'ont probablement jamais écrit beaucoup de programmes corrects.

0.2 Pourquoi SCILAB ?

Le choix d'un langage de programmation pour une initiation à l'informatique n'est pas, comme on l'a déjà laissé entendre, entièrement anodin. L'École des Mines de Saint-Etienne, qui ne répugne à aucune expérience dans le domaine, a utilisé pour ce faire des langages aussi variés que Fortran, Algol 60, APL, Pascal, Scheme, ADA, C, C++, et probablement quelques autres encore. Qu'ils soient impératifs, fonctionnels, à objets ou à enzymes gloutons[1], ces langages ont leurs avantages et leurs inconvénients, leurs partisans et leurs détracteurs, et ont fait parmi les étudiants des heureux, des insatisfaits, et dans tous les cas beaucoup d'indifférents. Sans déroger à la tradition (qui veut que tout cours d'informatique, péniblement mis au point sur une période de deux ou trois ans, soit pour des raisons totalement irrationnelles remplacé par autre chose de complètement différent), l'idée était ici de satisfaire tout autant les fantasmes des informaticiens que ceux des mathématiciens et physiciens en présentant aux étudiants un langage qu'ils puissent directement utiliser dans les autres cours.

[1] Terme de remplissage, utilisé ici en attendant la prochaine révolution...

0.2.1 Présentation de l'outil SCILAB.

SCILAB n'est pas à proprement parler un langage de programmation, mais plutôt un système complet, intégrant son propre langage (que nous nommerons également, par extension, SCILAB), des outils de mise au point, ainsi que diverses *bibliothèques* (ensemble de programmes applicatifs) essentiellement orientées vers les mathématiques et la physique.

SCILAB n'est pas non plus, à proprement parler, un *bon* langage de programmation[2], mais l'espoir de l'auteur de ce document est qu'il puisse néanmoins jouer son rôle de langage introductif à certains concepts de l'informatique.

Venons en maintenant à ce qu'*est* SCILAB. SCILAB est un logiciel de calcul numérique développé et distribué librement par l'INRIA, Institut National de Recherche en Informatique et Automatique. Les spécifications de SCILAB sont très proches de celles de MATLAB, auquel il constitue une alternative intéressante. Répondre à la question "pourquoi SCILAB ?" revient donc, d'une certaine manière, à répondre à la question "pourquoi MATLAB ?"

0.2.2 Pourquoi Matlab ?

L'outil MATLAB est un logiciel de calcul numérique très répandu dans le monde. Il est abondamment utilisé à l'École Nationale Supérieure des Mines de Saint-Etienne comme outil pédagogique : dans le pôle *Modélisation Mathématique*, dans les axes de deuxième année intitulés *Procédés et Systèmes Industriels*, *Instrumentation*, *Méthodes Statistiques et Prévision*, et enfin au sein de l'option *Procédés* en troisième année, ainsi que dans certains laboratoires de l'École.

MATLAB (ou en l'occurrence, SCILAB), fournissent les principales méthodes du calcul numérique :

- résolution de systèmes linéaires,
- calculs de valeurs et vecteurs propres,
- décomposition en valeurs singulières, pseudo-inverses,
- plusieurs méthodes de résolution d'équations différentielles,
- résolution d'équations non linéaires,
- plusieurs algorithmes d'optimisation,
- transformée de Fournier rapide,
- génération de nombres pseudo-aléatoires,
- ainsi que de nombreuses primitives d'algèbre linéaire, utiles par exemple en automatisme.

MATLAB propose un éventail assez complet de primitives graphiques, permettant la visualisation des résultats (tableaux, courbes, surfaces). Par ailleurs, de nombreux *packages*[3] sont disponibles, qui augmentent encore l'intérêt pratique de l'outil.

L'intérêt d'un apprentissage sérieux de MATLAB est aussi de pouvoir disposer en situation professionnelle de cet outil à la fois souple et puissant. Il nous a donc semblé utile d'en présenter les concepts fondamentaux le plus tôt possible dans le cursus ICM[4]. Comme indiqué, cette présentation s'effectue à travers SCILAB, qui est raisonnablement proche de MATLAB, et

[2] Le lecteur pourra s'en convaincre en jetant un coup d'oeil au bétisier, page **??**. D'un autre coté, les avis étant plus que divergents sur la question, il serait difficile de citer un langage qui soit universellement reconnu comme meilleur que SCILAB...

[3] Ensemble de programmes consacrés à la résolution de problèmes d'un domaine particulier.

[4] ICM : Ingénieur Civil des Mines

présente l'avantage d'être disponible sous la forme d'un logiciel libre (MATLAB restant coûteux, même dans le cas d'une utilisation universitaire). Nous signalerons à l'occasion les principales différences entre SCILAB et MATLAB à la fin de chaque chapitre. Le but de cette introduction étant, modestement, de faire assimiler aux étudiants les notions de base de programmation, le pari est fait que ces notions de base peuvent être transparentes au LAB utilisé (si, si !).

0.2.3 Quelques références

Les informations relatives à SCILAB sont disponibles à partir de la page Web du groupe SCILAB de l'INRIA :

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

On peut aussi se reporter à la page Web consacrée à SCILAB de l'auteur de ce document :

`http ://kiwi.emse.fr/SCILAB/`

La bibliographie de ce cours propose également quelques références papier concernant le langage. Il existe un petit nombre d'ouvrages tels que [7], ainsi que plusieurs documents internet consacrés à SCILAB. On citera en particulier les documentations du groupe SCILAB de l'INRIA (Introduction au langage [8]), et différents manuels et supports de cours, en français ([16]) ou en anglais ([21]).

Par ailleurs, nombre d'ouvrages sont consacrés à MATLAB, à commencer par le manuel de référence [20]. Citons aussi [15], et en français [14], [18].

0.3 Déroulement du cours

Ce cours est organisé dans le temps sous la forme de six demi-journées consécutives, suivies de quatre autres demi-journées réparties sur une durée d'un mois. Ces dernières séances permettent d'aborder avec SCILAB (ou MATLAB) les premières séances des cours de mathématiques/statistiques.

0.4 A propos de ce document

0.4.1 Finalité

Le but de ce document n'est pas de présenter toutes les fonctions du langage, loin de là. Il vise au contraire à présenter au lecteur les concepts généraux de l'informatique, en mettant ceux-ci en œuvre grâce à SCILAB (ou MATLAB), et à lui donner quelques indications sur la méthodologie d'écriture de programmes avec ces outils.

Cependant, l'apprentissage de la programmation n'est pas une chose simple, qui coule de source. C'est un objectif de longue haleine, qui nécessite nombre d'allers et retours (au travers des supports de cours, des exercices, des programmes, les vôtres, tout comme ceux qui vous sont donnés en exemple), d'autant plus que nous nous fixons deux buts différents, et presque, croyez-moi, contradictoires : l'apprentissage de la programmation, et l'apprentissage d'*un* langage de programmation...

0.4.2 Structure

Comme nombre d'ouvrages d'épaisseur suffisante pour passer pour un livre, celui-ci est structuré en chapitres, chaque chapitre correspondant, dans un monde idéal, à une séance de cours. Le premier de ces chapitres est une brève introduction à l'outil lui-même. Il faut bien, après tout, savoir le mettre en œuvre si l'on désire s'en servir. Le chapitre 2 est une introduction, assez traditionnelle dans son esprit, au langage SCILAB et au traitement des scalaires. Le chapitre 3 est une introduction à l'algorithmique, illustrée par de nombreux exemples programmés avec SCILAB. Le chapitre 4, aborde enfin le cœur du sujet, en introduisant les tableaux ainsi et leurs fonctions et algorithmes spécifiques à SCILAB. La suite du document, tout en présentant de nouvelles notions (chaînes de caractères, au chapitre **??**, listes, au chapitre **??**), vise essentiellement à conforter les anciennes, tant il est vrai que les multiples répétitions, sous diverses formes, facilitent l'apprentissage. Enfin, à côté de diverses annexes, ce document est doté d'un bétisier. Vous n'en profiterez réellement qu'à l'issue du cours. J'espère que sa lecture vous en paraîtra roborative, et, accessoirement, sera l'occasion de comprendre certains points de détail du langage et de son implantation.

Notons enfin que, pour des raisons pratiques liées au manque de temps du rédacteur, le document lui-même est partagé en deux parties. Les chapitres 1 à 6, qui correspondent au cours de base de la semaine bloquée de septembre, constituent un premier tome. Les chapitres 7 à 10, qui présentent des aspects plus avancés du langage et sont enseignés en octobre, ainsi que quelques annexes, constitueront le second tome.

0.4.3 Conventions

Les conventions utilisées dans ce documents sont assez intuitives (les fantaisies graphiques étant de toute façon limitées par le système de production utilisé) :

- le texte du cours utilise la police traditionnelle quasi imposée par TEX et LATEX ;
- les exemples sont imprimées avec une police de type Courrier : `police courrier` ;
- les formules mathématiques sont, chaque fois que c'est possible, imprimées selon les conventions chères aux mathématiciens. Un exemple que j'aime bien, mais que je ne suis pas sûr de parvenir à replacer ailleurs dans le document : $e^{i\pi} = -1$.
- une notation telle que FILE ▷ QUIT représente la sélection de l'item QUIT du menu FILE ;
- l'écriture Ctrl-B indique que vous appuyez sur la touche Ctrl du clavier en tapant la touche B (on peut faire ceci, au choix, avec deux mains ou une seule).
- une *emphase*, parfois ***lourdement*** appuyée, est mise sur les phases ou les mots *importants* (comme dans tout bon roman populaire, quand *ça va loin*, *voire même* ***très*** *loin*).

0.4.4 Réalisation

Ce document a été réalisé en juillet et août 2000, sur un PC portable opérant sous Linux. Le document a été saisi au moyen de LYX ([19]), un éditeur de documents qui permet d'utiliser LATEX ([17]) dans un mode quasi WYSIWYG.

0.4.5 Distribution

Ce document peut être librement distribué, utilisé, voire même *snarfé*[5], sous forme papier ou électronique. Pour toute demande spécifique, remarque ou correction, prendre contact avec l'auteur :

Jean-Jacques Girardot
École des Mines
158 Cours Fauriel
42023 Saint-Etienne Cedex, France
email : `girardot@emse.fr`

0.5 Remerciements

Je tiens à remercier tous ceux qui ont contribué de près ou de loin à la conception, la réalisation ou la mise en place de ce cours d'introduction, à commencer par les concepteurs de Scilab, à l'INRIA, les concepteurs de LyX [19], l'éditeur de documents utilisé pour rédiger ce cours, ainsi que mes collègues : Philippe Jaillon en premier, qui a débroussaillé les problèmes de mise en œuvre des divers logiciels, Yves Deville, auteur au sein de l'ENSMSE d'un précédent fascicule consacré à Matlab, Annie Corbel, qui a lu et relu avec beaucoup de mérite les versions successives de ce document, et de nombreux autres... Sans parler de tous ceux que j'ai abondamment *snarfés*, et qui ont ainsi largement contribué à la qualité de cet ouvrage.

[5] *snarfer* : *vt*, de *to snarf*, s'emparer, consciemment ou non, d'un texte, d'un fichier, d'une citation, d'un concept, etc., pour le réutiliser, avec ou sans l'accord de son légitime propriétaire ; explication snarfée de [1] (ou en français, [2]).

Première partie

Éléments de langage

Chapitre 1

Débuter avec SCILAB

Ce chapitre constitue un premier contact avec SCILAB.

1.1 Tutorial

1.1.1 Mise en œuvre

Le logiciel se lance par la commande `scilab` sous UNIX ou Linux, ou encore en cliquant sur l'icône correspondante sous d'autres systèmes, tels Windows ou MacOS. Le logiciel crée une fenêtre nouvelle (c.f. figure 1.1, page 18, pour l'apparence de cette fenêtre sous Linux, ou encore la figure 1.2, page 18, sous Windows).

Dans cette fenêtre, on trouve le message d'accueil du système :

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

                     Scilab-2.5
              Copyright (C) 1989-99 INRIA

    Startup exécution :
      loading initial environment

    -->
```

La suite de caractères "`-->`" imprimée par le logiciel est le *prompt*. Elle signale que SCILAB est en attente d'une commande de l'utilisateur. SCILAB est un *interprète* (c'est à dire que chaque commande introduite par l'utilisateur est immédiatement exécutée - une autre technique classique d'exécution des programmes est la *compilation*, dans laquelle un programme complet est analysé, transformé en un langage directement exécutable par la machine, puis effectivement exécuté).

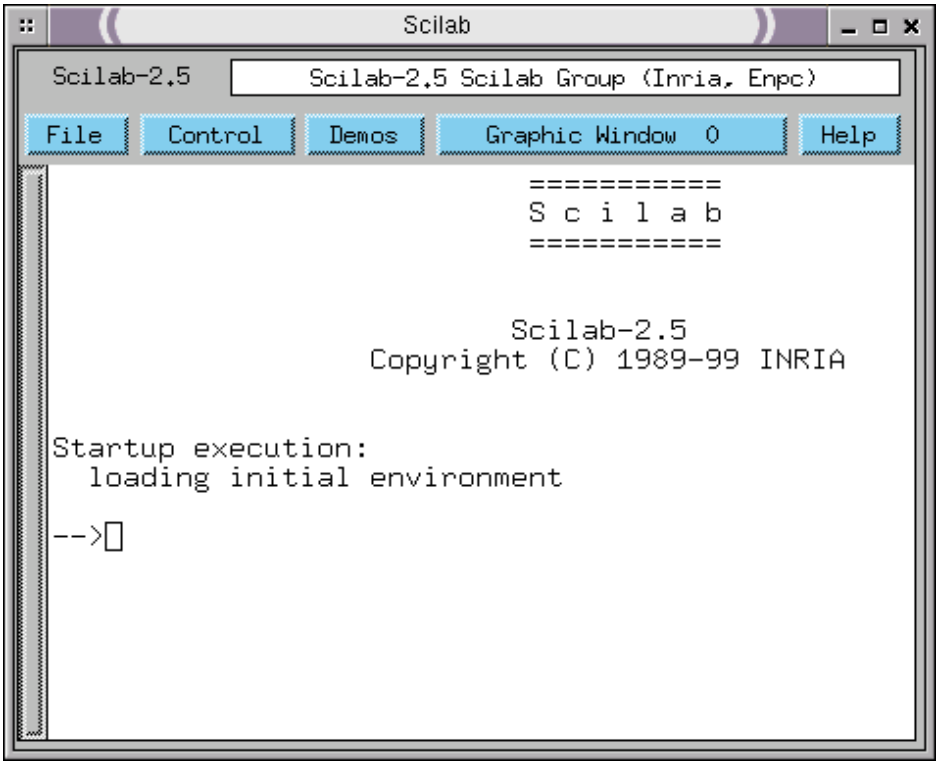

FIG. 1.1 – Fenêtre SCILAB sous Linux

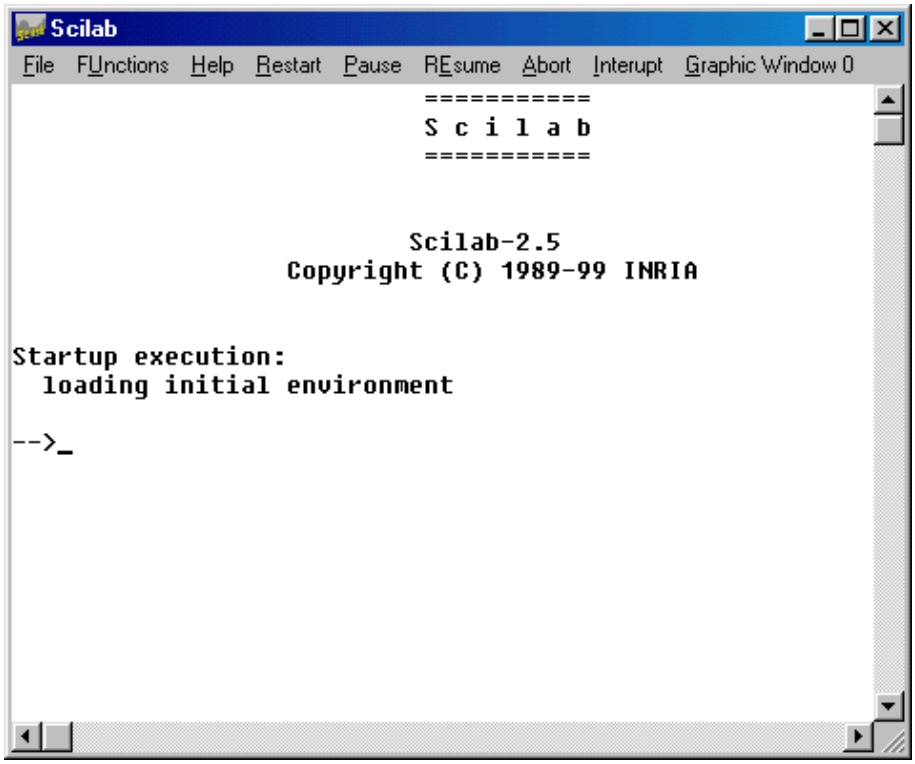

FIG. 1.2 – Fenêtre SCILAB sous Windows

```
--> a = 2 + 3
a =
  5.
```

L'utilisateur tape ensuite l'expression "`a = 2 + 3`" suivie de "RETURN" (sur certains claviers, cette touche est libellée "ENTER", ou marquée d'une simple flèche vers la gauche). La réponse du système consiste en la ligne "`a =`", suivie de la valeur de l'expression introduite. Le résultat est conservé dans la variable de nom `a`. On peut se contenter d'écrire :

```
--> 2 + 3
ans =
  5.
```

Comme précédemment, l'expression a été calculée. L'utilisateur n'ayant pas spécifié de destination pour la valeur de l'expression, le résultat de l'exécution de celle-ci est, par défaut, affecté à la variable `ans` (pour *answer*, bien sûr). Cette variable peut ultérieurement être utilisée dans toute expression :

```
--> ans + 3
 ans =
   8.
```

Par défaut également, SCILAB imprime la valeur calculée. Il est possible de supprimer cette impression en faisant suivre l'expression du caractère ; *(point-virgule)*. Ce caractère permet également d'introduire plusieurs expressions sur une même ligne (et agit donc comme *séparateur d'expressions*) :

```
-->2+3 ; ans+4 ; a=ans+10
 a =
   19.
```

Dans ce dernier exemple, le calcul de `2+3` affecte implicitement la valeur `5` à `ans` ; le calcul ultérieur, `ans+4`, donne la valeur finale `9` à la variable `ans`. L'expression affecte également la valeur `19.0` à la variable `a` au moyen de l'opération d'*affectation*, `=`.

La *virgule* peut également servir de *séparateur d'expressions*. Les résultats des différentes expressions sont alors imprimées :

```
-->a=5, b=3
 a =
   5.
 b =
   3.
```

On peut interrompre la session K par la commande `exit,` ou par le menu FILE ▷ QUIT ou FILE ▷ EXIT, selon les systèmes :

```
-->exit
```

Cette commande termine la session et ferme la fenêtre d'interaction avec SCILAB. Le mode d'interaction que nous avons utilisé ici est dit *mode terminal.* Nous verrons ultérieurement qu'il est possible de créer des fichiers contenant des instructions destinées à SCILAB, et de faire exécuter ces fichiers.

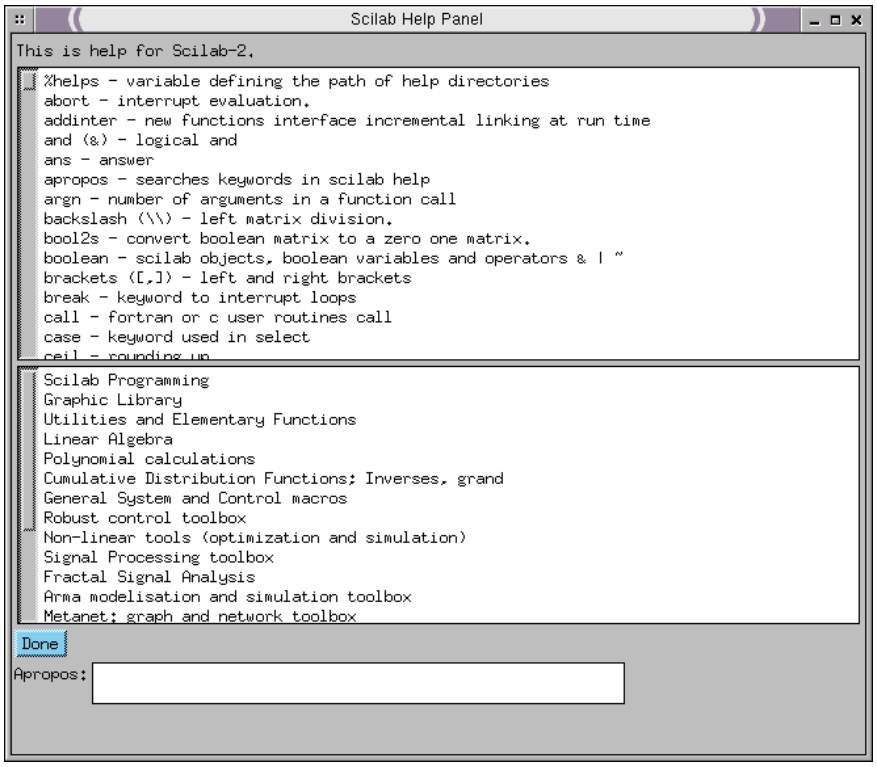

FIG. 1.3 – Fenêtre d'aide de SCILAB sous Linux

1.1.2 L'interface

La fenêtre SCILAB propose divers menus. La sélection FILE ▷ QUIT (que l'on obtient en cliquant sur le menu FILE, puis sur l'élément QUIT de ce menu), permet de quitter le logiciel.

Le menu DEMOS est amusant (mais à mon sens peu instructif).

On utilisera par contre fortement le menu HELP, qui fait apparaître une nouvelle fenêtre (c.f. figure 1.3, page 20, sous Linux, ou encore la figure 1.4, page 1.4, sous Windows), proposant une aide en trois temps. Le panneau du bas indique une catégorie, SCILAB PROGRAMMING étant présélectionnée. Le panneau du haut est la liste des items de cette catégorie. Cliquer sur l'un de ces éléments fait apparaître une nouvelle fenêtre, contenant le texte associé à cet item.

Une fonction similaire est réalisée par la commande en ligne `help`. Chacune des commandes ci-dessous fait apparaître la fenêtre d'aide liée au *mot-clef* spécifié.

```
-->help scilab
-->help help
-->help cos
-->help plus
-->help symbols
-->help keyboard
```

Les mots-clefs `scilab` et `help` nous renseignent respectivement sur l'utilisation du logiciel lui-même et sur la commande d'aide. Fort logiquement, `help cos` nous propose la description de l'opération primitive `cos` calculant le *cosinus* de son argument. `plus` est le mot-clef correspondant

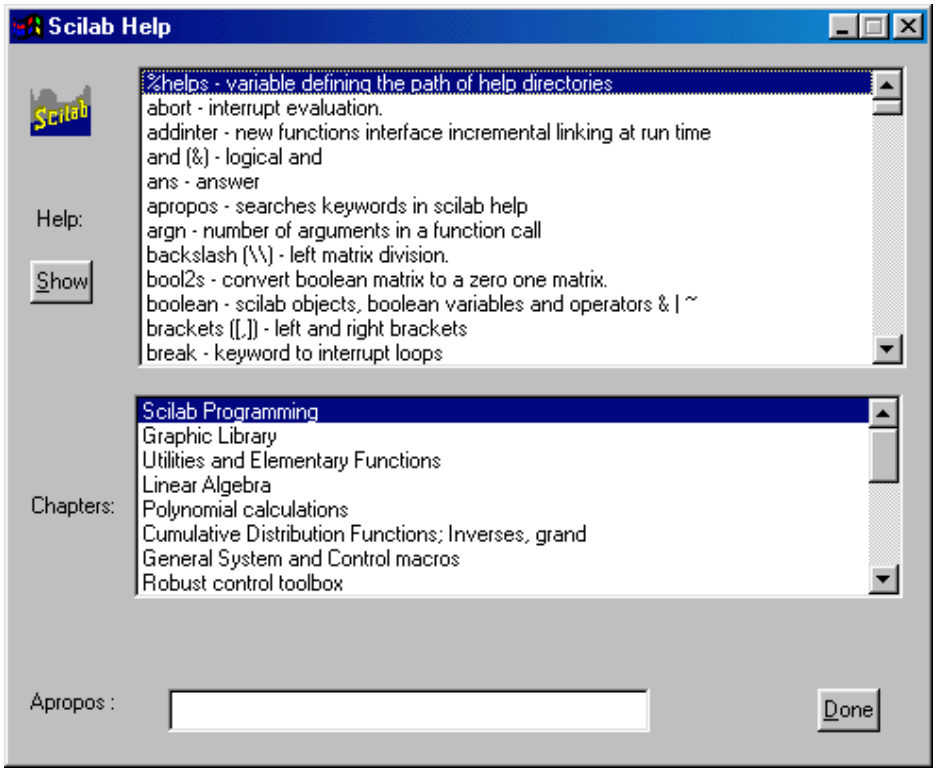

FIG. 1.4 – Fenêtre d'aide de SCILAB sous Windows

à l'opération d'addition, représentée par le caractère +. La fenêtre correspondant à l'affichage obtenu est donnée comme exemple à la figure 1.5, page 22. La liste des *symboles* ésotériques utilisés par SCILAB, ainsi que les mots-clefs qui leur sont associés, nous est donnée par `help symbols`. `help keyboard` fournit les équivalents clavier de certaines commandes.

Citons enfin la commande `apropos` ;

```
-->apropos cosin
```

Cette commande ouvre une fenêtre (c.f. figure 1.6, page 22) contenant la liste des fonctions du langage pour lesquelles le mot (ou en l'occurrence la sous-chaîne) "`cosin`" apparaît dans la description brève de la fonction. Là aussi, il suffit de cliquer sur l'une de ces lignes pour faire apparaître la fenêtre d'aide correspondante.

1.1.3 Ligne courante

La dernière ligne imprimée par SCILAB, celle qui débute par le prompt, est la *ligne courante*. C'est sur cette ligne que vous tapez les instructions destinées au logiciel. Vous pouvez vous déplacer horizontalement dans cette ligne (pour rajouter un espace, une parenthèse) au moyen des flèches horizontales du clavier, ou encore en utilisant Ctrl-B ou Ctrl-F (pour *backward* et *forward*). Les touches *delete* ou *backspace* permettent de supprimer un caractère de la ligne, etc.

SCILAB conserve un *historique* de la session (c'est à dire l'ensemble des commandes que vous avez tapées au cours de cette session), et vous pouvez rappeler une ligne antérieure, pour la ré-exécuter, après modification éventuelle. On utilise pour ceci les flèches verticales, ou encore

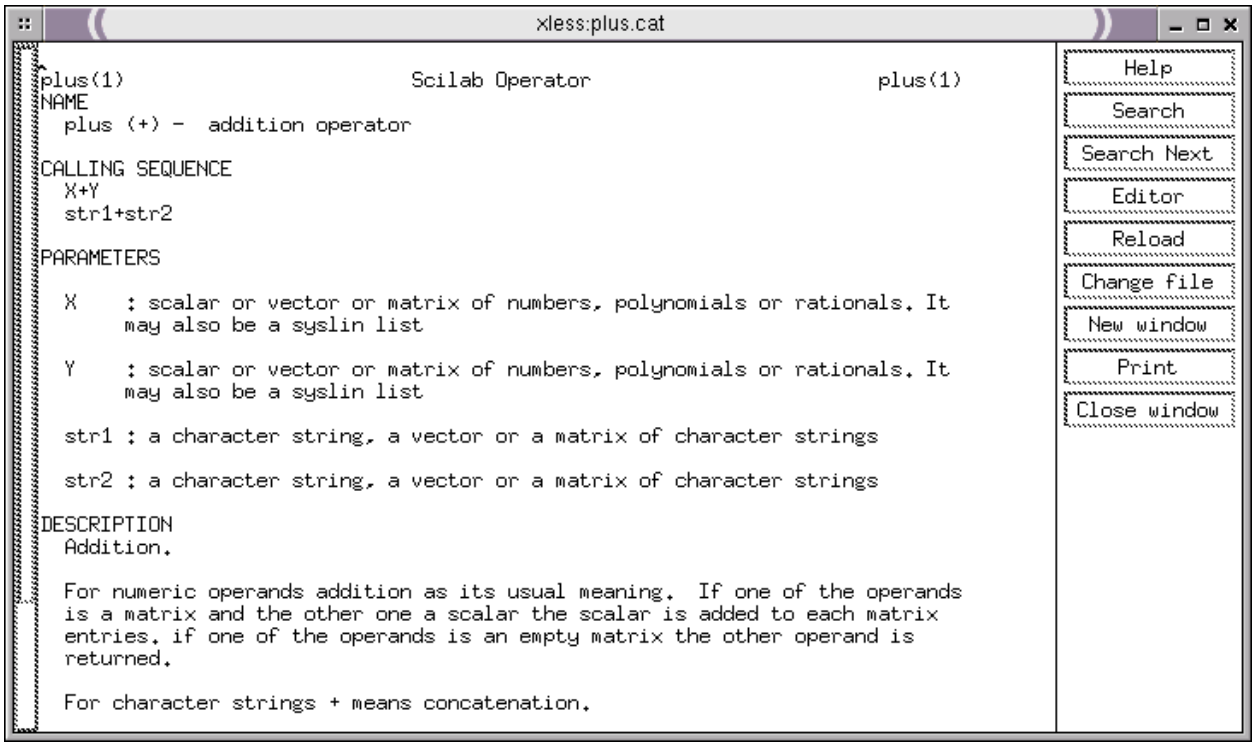

FIG. 1.5 – Fenêtre d'aide pour le mot-clef `plus`

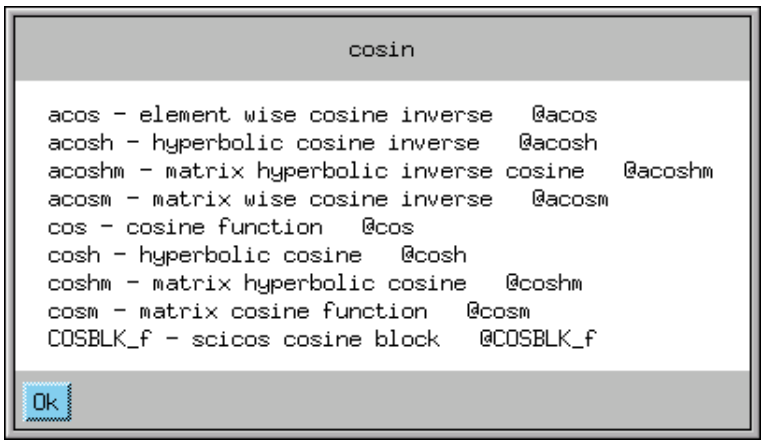

FIG. 1.6 – Fenêtre "`apropos cosin`"

Ctrl-P et Ctrl-N (pour *previous* et *next*), qui permettent de se "déplacer" verticalement d'une ligne à la fois. Mentionnons encore le point d'exclamation, qui, suivi de quelques caractères, rappelle la ligne débutant par ces caractères.

Enfin, il est possible d'introduire des commandes nécessitant plusieurs lignes pour leur écriture. On indique, en utilisant trois points consécutifs en fin de ligne, que la commande en cours se poursuit à la ligne suivante :

```
-->sin(0.195243)*sin(0.195243) + cos(0.195243) ...
--> * cos(0.195243)
 ans =
     1.
```

1.2 Variables

SCILAB permet de définir des *variables*, qui peuvent être *locales* à une procédure, résider dans l'*espace de travail*, ou encore être *globales*. Nous aborderons ultérieurement le cas des variables *locales* et *globales*, et nous ne nous intéresserons pour l'instant qu'aux variables de l'espace de travail.

Une *variable* est une *association* entre un *nom* et une *valeur*. Ainsi, la variable `ans` représente en SCILAB la dernière valeur calculée si ce résultat n'a pas été affecté à une autre variable.

1.2.1 Noms

Un *nom* est représenté par une suite de caractères, dont le premier est une lettre ou l'un des six caractères `% _ # ! $ ?`. Le reste d'un nom peut être composé de lettres, chiffres, ou des caractères `_ # ! $ ?`. En général, on évitera, pour des raisons de lisibilité, d'utiliser ces caractères. Les lettres majuscules et minuscules sont considérées comme différentes, et `toto` représente donc une autre variable que `Toto`. Un nom peut être aussi long que désiré, mais seuls les 24 premiers caractères sont pris en compte. Ainsi, les noms `Indicateur_ouverture_fichier_clients` et `Indicateur_ouverture_fichier_fournisseurs` représentent en fait la même variable[1].

1.2.2 Valeurs

Les *valeurs* que sait manipuler SCILAB peuvent être des valeurs numériques, des chaînes de caractères, ou des structures plus complexes. Nous reviendrons sur les chaînes de caractères au chapitre **??**, consacré au sujet.

1.2.2.1 Valeurs numériques

SCILAB gère un seul type de donnée numérique, qui est le type *complexe*. Un *complexe* est un assemblage de deux nombres dits *réels*[2] (valeurs qui représentent les parties réelle et imaginaire du nombre), qui s'introduisent sous une forme décimale. SCILAB permet des écritures telles que

[1] Mais qui s'amuserait à gérer des fichiers de clients ou de fournisseurs en Scilab ?

[2] Qui sont en réalité des fractions rationnelles, de la forme $N/2^P$.

`23.` ou `.5` pour représenter les nombres `23.0` et `0.5` (écritures dans lesquelles le *point décimal* est représenté par le caractère "."). Il est cependant préférable de systématiquement compléter ces écritures par un 0 (Le point est en effet également utilisé pour d'autres notations, ce qui peut être la cause d'erreurs subtiles).

Une valeur complexe s'obtient comme résultat d'un calcul :

```
-->sqrt(-4)
 ans   =
    2.i
```

Attention ! ***Une telle écriture n'est pas une notation acceptée pour un nombre complexe***.

```
-->2.i
   !--error 40
waiting for end of command[3]
```

Il est nécessaire d'écrire, par exemple :

```
-->2+3*sqrt(-1)
 ans   =
    2. + 3.i
```

La valeur obtenue est le *résultat* de la somme de 2 et du produit de 3 par le résultat de l'appel de la fonction `sqrt` avec -1 comme paramètre ; fonction qui, on l'a deviné, calcule la *racine carrée* de son argument[4]. On notera à cette occasion que les fonctions de SCILAB utilisent une syntaxe d'appel dans laquelle le nom de la fonction est suivi du paramètre de la fonction, placé entre parenthèses.

On peut aussi utiliser la variable prédéfinie `%i`, qui représente le nombre imaginaire i :

```
-->2+3*%i
 ans   =
    2. + 3.i
```

Les très grands et très petits nombres peuvent être représentés au moyen de la *notation exponentielle*. Une notation telle que `3.25E-21` représente le nombre $3,25 \times 10^{-21}$.

1.2.2.2 Valeurs spéciales

Citons pour mémoire deux valeurs particulières, qui peuvent s'avérer utiles dans certains cas, l'*infini* et l'*indéterminé*. Ces deux valeurs sont accessibles dans les variables prédéfinies `%inf` et `%nan`. Elles sont générées par certains calculs, et peuvent être testées par les opérations `isinf` et `isnan`.

[3] Notre première erreur !

[4] Le lecteur attentif aura naturellement remarqué que + représente l'addition, et * la multiplication...

1.2.3 Manipulation des variables

Les variables sont créées *dynamiquement* ; contrairement à ce qui se passe dans d'autres langages de programmation, elles n'ont pas besoin d'être déclarées. La *création* s'effectue au moyen de l'*opérateur d'affectation,* représenté par le signe `=`. Toute nouvelle affectation à une variable fait disparaître la valeur antérieure.

L'opération SCILAB `who` permet de connaître la liste des variables existantes.

```
-->who
your variables are...
b          a          startup    ierr       MODE_X     scicos_pal
%helps     MSDOS      home       PWD        TMPDIR     percentlib
fraclablib            soundlib   xdesslib   utillib    tdcslib    siglib
s2flib     roblib     optlib     metalib    elemlib    commlib    polylib
autolib    armalib    alglib     intlib     mtlblib    SCI        %F
%T         %z         %s         %nan       %inf       old
newstacksize          $          %t         %f         %eps       %io
%i         %e
using      5067 elements out of    1000000.
   and       47 variables out of   1071
```

Dans ce cas précis, seuls les deux premiers éléments de la liste, les variables `a` et `b`, ont effectivement été créées par l'utilisateur. Les autres sont des *variables prédéfinies* du système SCILAB, dont le rôle nous importe peu pour l'instant.

1.3 Différences entre Scilab et Matlab

Bien que SCILAB ait été conçu comme un clone de MATLAB, les logiciels présentent entre eux des différences plus ou moins importantes. L'utilisation d'un fichier de *start-up*, `.scilab`, placé dans le répertoire d'accueil de l'utilisateur et contenant les définitions appropriées, permet de contourner certaines des différences. Un tel fichier est listé en annexe (c.f. page 117).

1.3.1 Interface utilisateur

SCILAB et MATLAB proposent des interfaces utilisateur assez similaires. L'utilisateur familier de l'un de ces logiciels éprouvera peu de difficultés à passer à l'autre. Nous ne décrirons pas en détails les différences qui existent entre ces interfaces, en laissant l'utilisateur se reporter aux manuels de référence correspondants. On trouvera figure 1.7, page 26, l'aspect de la fenêtre MATLAB sous Windows, et figure 1.8, page 26, celui de la fenêtre d'aide.

1.3.2 Variables prédéfinies

Certaines *variables prédéfinies* sont dénotées de manières différentes en SCILAB et MATLAB. Par exemple, SCILAB utilise `%pi`, `%i` ou `%eps` pour dénoter les valeurs de π, du nombre imaginaire

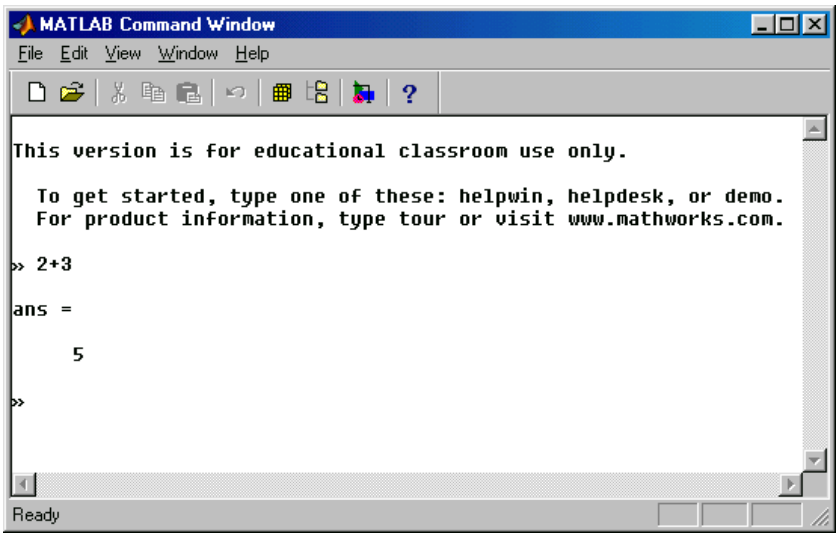

FIG. 1.7 – Fenêtre Matlab sous Windows

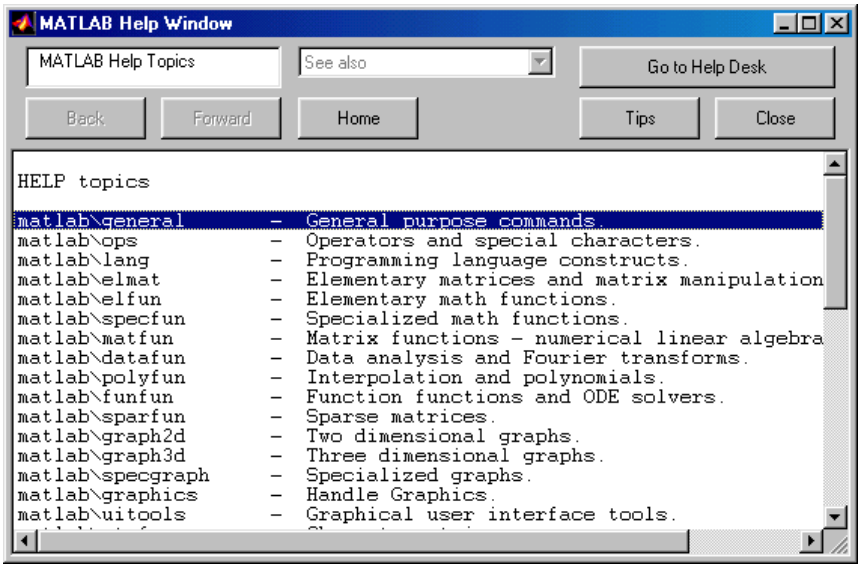

FIG. 1.8 – Fenêtre d'aide Matlab sous Windows

$\sqrt{-1}$, ou encore de la précision des calculs en virgule flottante, tandis que MATLAB notera `pi`, `i` et `eps` ces mêmes valeurs.

1.4 Exercices

Ces exercices ont pour but de vous permettre d'utiliser SCILAB comme d'une calculette, en même temps qu'ils vous incitent à consulter la documentation en ligne.

1.4.1 Hypoténuse

Soient `A=12` et `B=16` les longueurs des côtés de l'angle droit d'un triangle rectangle ; calculer la longueur de l'hypoténuse.

1.4.2 Disque

Quelle est la surface d'un disque de 12 cm de diamètre ?

1.4.3 Nombres complexes

Soit `C=3+4*%i` ; comment obtenir les parties réelles et imaginaires de ce nombre ? Comment en calculer ρ et θ, c'est à dire le *module* et l'*argument* ? Réciproquement, étant données ces deux dernières valeurs, comment obtenir le nombre complexe correspondant ?

1.4.4 Affichage

Calculez `N=1/3` ; quelle valeur s'affiche pour `N` ? Recopiez cette valeur, multipliez-la par `3`. Quel est le résultat ? Calculez maintenant `N*3` ; qu'en déduisez-vous ?

1.4.5 Approximations

Soient les nombres `X=1E30`, `Y=1E10` ; calculer `X+Y-X`, puis `X-X+Y`. Qu'en déduisez-vous ? Si l'on choisit `Y=1`, quelle est la valeur minimale de `X` pour laquelle de phénomène se produit ?

1.4.6 Tracé de courbes

Chercher dans la documentation comment tracer le graphe de la fonction *sinus* entre 0 et $2 \times \pi$.

1.4.7 Interface

Que se passe-t-il quand vous utilisez un `!` en début de ligne ?

1.4.8 Programme nu (pour les utilisateur de SCILAB sous LINUX ou UNIX)

Comment faire pour lancer SCILAB sans que celui-ci n'ouvre une fenêtre nouvelle ? A votre avis, pourquoi ferait-on une chose pareille ?

1.4.9 Menus

Ajouter à la fenêtre SCILAB un nouveau menu, "SALUT", avec les deux items "BONJOUR" et "AU REVOIR" ; les choix de ces menus entraîneront les impressions respectives de "HELLO" et "GOOD BYE".

1.4.10 Compatibilité

Créer le fichier `.scilab` ; y définir ce qu'il faut pour que i et pi aient des valeurs convenables lorsque vous lancez votre version du logiciel.

Chapitre 2

SCILAB, le langage

2.1 Introduction

Comme tout langage de programmation, SCILAB permet de transcrire des algorithmes manipulant valeurs, variables, et faisant appel à des fonctions primitives ou définies. Comme dans la plupart des langages de programmation, tels *C* ou *Pascal*, cette manipulation peut s'effectuer à un niveau *scalaire*, c'est à dire élément par élément. C'est ce que nous allons aborder dans ce chapitre. SCILAB offre également, ce qui est plus original et fait sa force, des opérations vectorielles ou matricielles ; nous aborderons cet aspect dans le chapitre 4, consacré aux tableaux.

2.2 Éléments syntaxiques

2.2.1 Objets de base

Les objets de base du langage sont les constantes (nombres et chaînes de caractères) et les variables, déjà présentées au chapitre précédent.

2.2.2 Expressions

Les expressions du langage calculent des valeurs par application d'opérations sur les objets de base, constantes et variables. Nous nous limiterons dans ce chapitre aux opérations s'appliquant aux scalaires. Certaines opérations du langage sont représentées par des symboles particulier (comme `+`, `>`, etc.), d'autres par des noms (comme `sin`, `cos`, `modulo`, etc.).

2.2.3 Opérations arithmétiques

Les opérations arithmétiques vont, comme leur nom le suggère, s'appliquer aux nombres. Nous verrons successivement celles qui sont représentées par des symboles, s'appliquant à un ou deux opérandes, puis celles qui sont représentées par des mots réservés du langage.

Opération	Description
-	*Opposé* de son paramètre
+	*Identité*

FIG. 2.1 – Opérateurs arithmétiques monadiques

Opération	Description
^ ou **	*Élévation à la puissance*
*	*Produit* de deux valeurs numériques
/	*Quotient* de deux valeurs numériques
\	*Quotient inversé* de deux valeurs numériques : A\B équivaut à B/A.
+	*Addition* de deux valeurs numériques
-	*Différence* de deux valeurs numériques

FIG. 2.2 – Opérations arithmétiques dyadiques usuelles

2.2.3.1 Opérateurs monadiques

Les opérateurs *monadiques* s'appliquent à un opérande, qu'ils peuvent précéder (opérateur *préfixe*), ou suivre (opérateur *infixe*). La figure 2.1, page 30, en donne la description. Quelques exemples :[1]

```
-->- 3
 ans = - 3.
-->+ 5
 ans =  5.
```

2.2.3.2 Opérateurs dyadiques

Les opérations arithmétiques usuelles sont disponibles sous la forme d'opérateurs *dyadiques*. Ces opérations s'appliquent à deux opérandes, entre lesquels est placé le symbole représentant l'opération ; on parle d'opérateurs *infixes*. La figure 2.2, page 30, en donne la description succincte. Dans ce tableau, les opérations sont rangées en trois classes de priorités décroissantes : l'élévation à la puissance est prioritaire sur les produits et quotients, eux-même prioritaires sur les additions et soustractions ; c'est ce que montre l'exemple suivant, dans lequel la seconde expression, entièrement parenthésée, fournit le même résultat que la première.

```
-->1+2^4*10
 ans  =  161.
-->1+((2^4)*10)
 ans  =  161.
```

[1] Nous simplifierons dorénavant l'écriture des résultats fournis par SCILAB, et économiserons du papier, en représentant sur la même ligne le nom de la variable affectée et la valeur de cette variable.

Opération	Description
`abs(x)`	*valeur absolue*, *magnitude*
`atan(x)`	*arc tangente*
`ceil(x)`	*partie entière*, arrondi vers $+\infty$
`conj(x)`	*conjugué* de son paramètre
`cos(x)`	*cosinus*
`exp(x)`	*exponentielle*
`fix(x)`	*partie entière*, arrondi vers 0
`floor(x)`	*partie entière*, arrondi vers $-\infty$
`imag(x)`	*partie imaginaire* d'un nombre complexe
`int(x)`	*partie entière*, arrondi vers 0 (idem. `fix`)
`log(x)`	*logarithme naturel*
`log10(x)`	*logarithme en base 10*
`modulo(x,y)`	*reste* de la division de `x` par `y`
`real(x)`	*partie réelle* d'un nombre complexe
`round(x)`	*partie entière*, arrondi vers l'entier le plus proche
`sin(x)`	*sinus*
`sqrt(x)`	*racine carrée*
`tan(x)`	*tangente*

FIG. 2.3 – Fonctions mathématiques usuelles de SCILAB

Pour un même niveau de priorité, les opérations sont réalisées dans un ordre prédéterminé (de gauche à droite pour les additions[2] et produits, de droite à gauche pour l'élévation à la puissance). Ainsi, `A-B+C` équivaut à `(A-B)+C`.

2.2.3.3 Fonctions

Arithmétique Le langage SCILAB comporte, de par sa vocation, toutes les fonctions arithmétiques usuelles, ainsi que de nombreuses autres. La table 2.3, page 31, énumère les plus traditionnelles. On trouvera aisément les autres par application des algorithmes de recherche appris au cours de la première séance. Quelques exemples :

```
-->modulo(13,5)
 ans =  3.
-->modulo(12.2,5.4)
 ans =  1.4
```

Dans le dernier exemple du `modulo`, la réponse, `1.4`, est égale à `12.2-x*5.4`, où `x` est le plus grand entier (ici, `2`) tel que `x*5.4` soit inférieur à `12.2`.

On notera que le langage fournit plusieurs opérations différentes donnant la partie entière d'un nombre, selon le mode d'arrondi désiré (`fix` ou `int`, `round`, `ceil` et `floor`).

Autres opérations utiles La table 2.4, page 2.4, présente quelques autres fonctions de SCI-

[2]Terme générique pour *additions et soustractions* ; de même, *produits* signifie *multiplications et divisions*.

Opération	Description
`disp(x,y...)`	*affichage* des valeurs des paramètres x, y, etc.
`format(n)`	sélection de n chiffres significatifs pour l'affichage des nombres
`string(x)`	*chaîne de caractères* représentant le paramètre
`rand()`	*nombre aléatoire* compris entre 0 et 1.

FIG. 2.4 – Quelques autres fonctions de SCILAB

Opération	Description
$<$	*Inférieur*
$<=$	*Inférieur ou égal*
$>$	*Supérieur*
$>=$	*Supérieur ou égal*
$==$	*Égal*
$<>$ ou $\tilde{}=$	*Différent*

FIG. 2.5 – Opérations de comparaison

LAB, qui s'avèreront utiles dans ce chapitre. Les chaînes de caractères seront abordées plus en détail dans un chapitre ultérieur, mais nous utiliserons à l'occasion la fonction `disp()` qui nous permettra d'imprimer un message, souvent associée à la fonction `string()`, qui convertit un nombre en une chaîne de caractères :

```
-->disp("Le produit de 2 et 3 vaut " + string(2*3))
Le produit de 2 et 3 vaut 6
```

Dans cet exemple, `string(2*3)` calcule l'expression `2*3`, et convertit le résultat en une chaîne de caractères ; le signe `+` qui précède est l'opération d'*addition de chaînes de caractères*, qui va placer bout à bout ses deux opérandes (on parle de *concaténation*). Le résultat, enfin, est imprimé par la fonction `disp()`.

Notons aussi la fonction `rand()`, qui fournit un nombre aléatoire compris entre 0 et 1. Les nombres sont uniformément répartis entre ces deux valeurs. Le paramètre de `rand()` n'est pas significatif dans cette utilisation, et peut être omis :

```
-->rand(0)
 ans = 0.2113249
-->rand()
 ans = 0.7560439
-->rand()
 ans = 0.0002211
```

2.2.4 Comparaisons

Les opérations de *comparaison* vont permettre de tester *l'égalité* de deux valeurs, et aussi, dans le cas de valeurs numériques, de comparer ces valeurs entre elles. Le tableau 2.5, page 32, donne la description sommaire des fonctions de comparaison.

Le résultat de ces opérations est une donnée d'une type nouveau, le *type logique*, dit aussi *type booléen* Une valeur logique s'imprime sous la forme des caractères `T`, pour *true,* valeur *vraie*, et `F`, pour *false*, valeur *fausse*. Quelques exemples :

```
-->4>3
 ans =  T
-->2>3
 ans =  F
-->2>3 == F
 ans =  T
-->2*3+1==4+3
 ans =  T
```

Les opérations de comparaison ont *même priorité*, et s'effectuent de gauche à droite. Dans l'avant-dernier exemple, le résultat de `2>3` est comparé à la valeur `F`, fournissant le résultat final `T`.

La priorité des opérations de comparaison est plus *faible* que celles des opérations arithmétiques, comme le montre le dernier exemple dans lequel les expressions `2*3+1` et `4+3` sont évaluées avant l'opération `==`.

Analysons enfin ces exemples faisant intervenir des nombres complexes :

```
-->i == sqrt(-1)
 ans =  T
-->2+3*i == 5
 ans =  F
-->2+3*i < 3
         !--error  4
undefined variable : %s_1_s
```

Assez logiquement, SCILAB peut déterminer si deux complexes sont égaux ou non ; en revanche, aucune relation d'ordre n'étant définie sur les complexes, l'opération `<` échoue en signalant une erreur[3].

2.2.5 Expressions logiques

Les expressions logiques, comme leur nom le laisse supposer, vont nous permettre de combiner entre elles des valeurs logiques, pour obtenir de nouvelles valeurs logiques. Les opérations logiques habituelles sont la *négation logique*, ou *not*, le *et*, le *ou inclusif* et le *ou exclusif*. La table 2.6, page 34, décrit ces différentes opérations. On notera qu'il n'existe pas vraiment d'opération spécifique pour représenter la notion de *ou exclusif*, mais qu'une fonction équivalente est fournie par l'opération de comparaison, `<>`.

[3]Le message cryptique "`undefined variable : %s_1_s`" indique en fait qu'il serait possible à l'utilisateur de définir une fonction permettant cette comparaison, mais qu'aucune n'est présentement définie ; le lecteur intéressé pourra consulter l'aide associée au mot "`overloading`".

Opération	Description
~	*Négation logique*, ou *Not*
&	*Et logique*
|	*Ou inclusif*
<>	*Ou exclusif*

FIG. 2.6 – Opérations logiques

Les priorités des opérateurs `~`, `&` et `|` sont décroissantes, c'est-à-dire qu'une expression telle que `A|~B&C`, dans laquelle `A`, `B` et `C` sont des variables logiques, est équivalente à l'expression parenthésée `A|((~B)&C)`. La priorité des opérations logiques est plus faible que celles des opérations de comparaison, ce qui permet d'éviter l'utilisation de parenthèses dans une expression telle que `x>1&x<10`, qui rend vrai si `x` est dans l'intervalle ouvert]1 10[. Naturellement, la priorité de `<>` reste celle d'une opération de comparaison, ce qui veut dire qu'il faudra utiliser des parenthèses autour de ses paramètres si ceux-ci sont des expressions arithmétiques ou de comparaison.

2.2.6 Priorités

La priorité des opérations logiques est inférieure à celle des opérations de comparaison, qui est elle-même inférieure à celle des opérations arithmétiques.

Les opérations *monadiques* `+` et `-` ont même priorité que leurs pendants *dyadiques*. Une expression telle que `-A-B` est donc équivalente à `(-A)-B`. De manière systématique, pour éviter toute ambiguïté, on placera entre parenthèses toute expression du type `-X` ayant pour but de calculer l'opposé de `X`.

2.2.7 Conversion implicite

Scilab pratique une *conversion implicite* des opérandes de type logique vers le type numérique lorsque rendu nécessaire par le contexte. Les valeurs *vrai* et *faux* sont alors converties en 1 et 0 respectivement :

```
-->3+T
 ans = 4.
-->F+0
 ans = 0.
-->F==0
 ans = F
-->F+0==0
 ans = T
```

Attention ! la troisième expression nous dit que *faux* n'est pas égal à zéro ; la quatrième calcule `F+0`, ce qui fournit 0, et compare ce résultat à 0, rendant *vrai*.

2.3 Instructions

2.3.1 Instruction simple

L'instruction la plus simple est celle qui comporte une expression unique. Cette instruction *doit se terminer* par un passage à la ligne, une virgule, ou un point-virgule. On a déjà signalé que dans ce dernier cas, le résultat de l'expression n'était pas imprimé sur le terminal, alors qu'il l'était dans les autres cas.

2.3.2 Instruction vide

Dans le cas particulier où une instruction simple ne comporte pas d'expression, elle est dite *vide*. Elle consiste donc en un passage à la ligne, une virgule, ou un point-virgule.

2.3.3 Affectation

L'affectation est l'instruction qui permet de donner à une variable une valeur qui est le résultat d'un calcul. L'opération est représentée par le symbole =. La forme la plus simple est :

nom = expression

Le signe = est dit "opérateur d'affectation".

Note Il n'est pas possible de réaffecter les variables prédéfinies de SCILAB, telles `%pi` ou `%i`, ni les fonctions primitives :

```
-->%pi=3
      !--error 13
redefining permanent variable
-->sin=3
      !--error 25
bad call to primitive :sin
```

2.3.4 Séquences d'instructions

Une séquence d'instructions se compose d'une ou plusieurs instructions (éventuellement vides), placées les unes après les autres. Ex :

```
a=2 ; b=3 ; a+b ;
```

Comme on l'a signalé à plusieurs reprises, ces instructions peuvent individuellement se terminer par un point-virgule, une virgule, ou un passage à la ligne.

2.3.5 Instructions conditionnelles

L'idée de l'instruction *conditionnelle* est de conditionner l'exécution d'une séquence d'instructions à la réussite (ou l'échec) d'une certaine condition.

2.3.5.1 L'instruction *if*

L'instruction `if` permet l'exécution conditionnelle de séquences d'instructions. Sa syntaxe la plus simple est :

```
if expr then insts end
```

Lorsque l'expression ***expr*** est vraie, la séquence d'instructions ***insts*** est exécutée. La syntaxe de l'expression fait intervenir les *mots-clefs*[4] `if`, `then` et `end`. On notera que les instructions de ***insts*** doivent se terminer par une virgule (auquel cas leur résultat sera imprimé) ou un point-virgule (il ne le sera pas) :

```
-->a=2 ; if a == 2 then 3+4, end
 ans   =  7.
-->a=5 ; if a == 2 then 3+4, end
-->
```

Dans le deuxième exemple, le résultat du test est *faux* ; l'addition n'est donc pas exécutée, et il n'y a pas d'impression du résultat.

Il est possible de préciser une instruction (ou une séquence d'instructions) à exécuter lorsque la condition testée est fausse. Cette instruction est introduite par le mot-clef `else`. La syntaxe de l'expression est alors la suivante :

```
if expr then inst1 else inst2 end
```

Lorsque l'expression ***expr*** est vraie, l'instruction ***inst1*** est exécutée ; dans le cas contraire, l'instruction ***inst2*** est exécutée.

Notes Le *mot-clef* `then` peut être remplacé par un passage à la ligne, une virgule ou un point-virgule. On évitera cette «facilité» qui rend encore plus obscurs les programmes...

Notons encore qu'il est possible de remplacer des imbrications d'instructions conditionnelles telles que :

```
if condition1 then
  actions1
else
  if condition2 then
    actions2
  else
    if condition3 then
      actions3
    else
      actions4
    end
  end
end
```

[4] Un *mot-clef* est un élément d'un langage de programmation utilisé pour délimiter une construction syntaxique particulière ; dans notre cas, l'instruction conditionnelle.

par la forme équivalente suivante :

```
if condition1 then
  actions1
elseif condition2 then
  actions2
elseif condition3 then
  actions3
else
  actions4
end
```

Cette dernière syntaxe utilise le mot-clef `elseif` et permet donc d'effectuer une série de tests successifs, auxquels sont associés des actions.

2.3.5.2 Exemples

Échange conditionnel des valeurs de deux variables Étant donné deux variables, `a` et `b`, il s'agit d'obtenir dans `a` la plus petite des deux valeurs, dans `b` la plus grande. Pour résoudre ce problème, on va comparer `a` et `b`, et échanger éventuellement leurs valeurs en passant par une variable intermédiaire, `c` :

```
if a>b then c=a ; a=b ; b=c ; end
```

On prétend parfois que l'instruction suivante effectue le même travail ; qu'en pensez-vous ?

```
if a>b then a=a+b ; b=a-b ; a=a-b ; end
```

2.3.6 L'instruction de sélection

L'*instruction de sélection* est très voisine de cette dernière forme d'instruction conditionnelle. Sa syntaxe est la suivante :

```
select expression0
  case expression1 then
    actions1
  case expression2 then
    actions2
  case expression3 then
    actions3
  else
    actions4
end
```

Elle fait appel aux mots-clefs `select`, `case`, `then`, `else` et `end`. La valeur d'*expression0* est calculée, puis comparée successivement aux valeurs d'*expression1* , *expression2* , etc. Dès que l'une de ces comparaisons rend *vrai*, l'action correspondante se termine, et les autres expressions

ne sont plus testées. Si aucune des comparaisons ne rend *vrai*, l'action par défaut, introduite par le mot-clef `else`, est exécutée. La construction est donc équivalente à la séquence suivante, dans laquelle on a introduit la variable `expr` pour conserver le résultat du calcul de *expression0* :

```
expr = expression0 ;
if expr == expression1 then
    actions1
elseif expr == expression2 then
    actions2
elseif expr == expression3 then
    actions3
else
    actions4
end
```

2.3.7 Instructions répétitives

Une *instruction répétitive* (on parle aussi de *boucle*) a pour but d'exécuter plusieurs fois en séquence une instruction, ou un groupe d'instructions. Il existe deux formes d'instructions répétitives, les boucles *for* et les boucles *while*.

2.3.7.1 L'instruction *for*

La boucle *for* permet de répéter un groupe d'instructions alors qu'une variable particulière, la *variable de boucle*, décrit un ensemble de valeurs. La syntaxe de l'instruction *for* est la suivante :

```
for variable=valeurs do instructions end
```

Cette notation fait intervenir les mots-clefs `for`, `do` et `end`. *variable* est le nom de la variable de contrôle de la boucle. *instructions* consiste en une ou plusieurs instructions (se terminant par des virgules ou des points-virgules, avec le sens habituel). *valeurs* est la liste des valeurs que va décrire la variable de contrôle de la boucle. De manière générale, cet ensemble de valeurs est défini par un vecteur ou une matrice (c.f. chapitre 4). Nous nous bornerons pour l'instant aux notations suivantes :

```
début : pas : fin
début : fin
```

dans lesquelles *début* représente la valeur initiale de la variable de boucle, *fin* une valeur de fin qui ne sera pas dépassée, et *pas* le pas de progression de la variable de boucle (la seconde notation est équivalente à la première, avec une valeur du *pas* égale à 1).

Voici un exemple d'utilisation de la boucle *for* pour calculer factorielle(10) :

```
-->a=1 ;
-->for n=1 :10 do
--> a=a*n ;
-->end
```

```
-->a
 a =  3628800.
```

De même, le calcul de $\sum_{i=1}^{i=10} \frac{1}{i}$ peut s'écrire :

```
-->somme=0 ;
-->for n=1 :10 do somme=somme+1/n ; end
-->somme
 somme =  2.9289683
```

Notes : Dans certains cas, la valeur finale peut ne pas être atteinte, si le pas utilisé est tel qu'une itération supplémentaire ferait dépasser à la variable de boucle cette valeur finale :

```
-->for k=1 :4 :10 do
-->    disp(k) ;
-->end
   1.
   5.
   9.
-->
```

Remarque : Le *mot-clef* `do` peut être remplacé par un passage à la ligne, une virgule ou un point-virgule. On évitera cette «facilité» qui rend encore plus obscurs les programmes...

On notera également que la variable de contrôle est strictement *locale* à la boucle *for*, et n'a pas d'existence à l'extérieur de celle-ci.

2.3.7.2 L'instruction *while*

Autre instruction répétitive, la boucle *while* va nous permettre de répéter un groupe d'instructions tant qu'une condition est réalisée. La syntaxe de l'instruction *while* est la suivante :

`while` ***expression*** `do` ***instructions*** `end`

Cette notation fait intervenir les mots-clefs `while`, `do` et `end`. ***expression*** doit fournir un résultat de type booléen. ***instructions*** consiste en une ou plusieurs instructions (se terminant par des virgules ou des points-virgules, avec le sens habituel).

Le calcul de la factorielle de 10 peut s'écrire ainsi au moyen d'une boucle *while* :

```
-->n=10 ;
-->a=1 ;
-->while n>0 do
-->  a=a*n ;
-->  n=n-1 ;
-->end
-->a
 a =  3628800.
```

Note Là encore, le *mot-clef* `do` peut être remplacé par `then`, par un passage à la ligne, une virgule ou un point-virgule. On évitera cette «facilité» qui rend encore plus obscurs les programmes...

2.3.7.3 L'instruction *while... else*

L'instruction while admet une autre forme, dont la syntaxe est la suivante :

`while` *expression* `do` *instructions* `else` *instructions* `end`

Cette notation fait intervenir le mot clef `else`. La *sémantique*[5] est la même que dans l'autre forme de l'instruction *while*, mais permet en outre de préciser une ou plusieurs instructions qui seront exécutées à la fin de la boucle while, quand la condition testée devient fausse :

```
-->i=10 ; while i>0 do
-->  i = i-3 ;
-->else
-->  i
-->end
i =   - 2.
```

2.3.7.4 L'instruction *break*

Utilisée à l'intérieur d'une boucle *for* ou *while*, l'instruction `break` permet d'interrompre la boucle, et de sauter immédiatement à l'instruction qui suit cette boucle dans le texte du programme en cours d'exécution. En reprenant l'exemple ci-dessus :

```
-->i=10 ; while i>0 do
-->  i = i-3 ;
-->  if i==7 then break ; end
-->else
-->  disp(i)
-->end
```

l'exécution du programme montre que la partie *else* de la boucle n'est pas exécutée lorsque l'instruction `break` est déclenchée.

2.4 Fonctions

2.4.1 Définition

L'un des principaux attraits de SCILAB est qu'il permet à l'utilisateur de créer de nouvelles fonctions. Les fonctions dont il est question ici ne sont pas tout-à-fait des fonctions au sens mathématiques du terme. On parle habituellement, en informatique, de *procédure*. Il s'agit d'implanter, en utilisant les instructions du langage, un algorithme qui sera désigné par un nom,

[5] C'est-à-dire la *signification* de la construction.

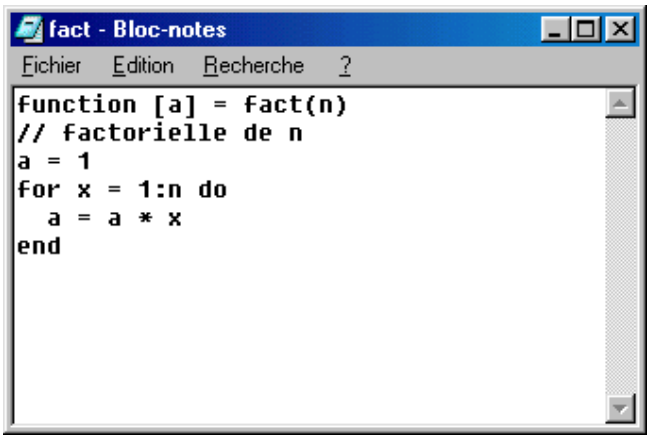

FIG. 2.7 – Bloc-notes Windows

pouvant avoir des entrées, des sorties, faire référence à des données externes et à d'autres procédures. On parlera en l'occurrence de *fonction définie*, par opposition avec les *fonctions primitives* du langage, telles `sin` ou `cos`.

On s'en tiendra pour l'instant aux règles d'or suivantes :

- chaque fonction définie est placée dans un *fichier*, dont le nom est celui de la fonction définie, et dont l'extension est `.sci` (d'où le terme de *sci-file* parfois utilisé pour caractériser ces fichiers) ;
- le fichier débute par la déclaration de fonction, indiquée par le mot-clef `function`, et dont la syntaxe est :

  `function` *résultats* `=` *nom paramètres*

- les commentaires éventuels (c.f. § 2.5.1.1) ne sont admis qu'après la ligne déclarant la fonction ;
- les instructions constituant le corps de la fonction suivent la déclaration de fonction ;
- le fichier se termine par une ligne vide ou blanche.

Une fonction peut avoir 0, 1 ou plusieurs *paramètres*. Ceux-ci sont dénotés par une liste de noms séparés par des virgules, placée entre *parenthèses*. La liste peut éventuellement être vide, et la déclaration des paramètres ne comporte alors qu'une paire de parenthèses.

Une fonction peut avoir 0, 1 ou plusieurs *résultats*. Ceux-ci sont dénotés par une liste de noms séparés par des virgules, placée entre *crochets*. La fonction peut ne pas comporter de résultat ; le mot-clef `function` sera alors immédiatement suivi du nom de la fonction et de ses paramètres. Si la fonction n'a qu'un résultat, les crochets sont facultatifs.

Sous Linux, un tel fichier peut être créé au moyen d'un éditeur de textes tel *emacs* ou *vi*, ou encore *gedit*, *gnotepad*, etc.

Sous Windows, le plus simple est d'utiliser le bloc-notes (c.f. fig. 2.7, page 41).

À titre d'exemple, voici le contenu du fichier `fact.sci` :

```
function [a] = fact(n)
// factorielle de n
a = 1
for x = 1 :n do
```

```
    a = a * x
end
```

Une fonction sera chargée en mémoire par l'opération `getf`. Si le fichier ci-dessus se trouve dans le répertoire `exos` de l'utilisateur, la commande de chargement sera :

```
getf('exos/fact.sci')
```

(Sous Windows, il est possible d'utiliser le menu FILE ▷ GETF, puis de sélectionner interactivement le fichier à charger.)

La fonction devient dès lors immédiatement disponible :

```
-->getf('exos/fact.sci')
-->fact(6)
 ans =  720.
```

Note : de manière générale, ***tout texte de programme cité dans ce cours, qui débute par le mot `function` doit avoir été créé sous forme d'un fichier texte (d'extension `.sci`) et chargé dans le logiciel au moyen de `getf`.***

2.4.2 Création de fonction

La procédure que nous venons de décrire est celle qui doit être ***systématiquement*** utilisée pour la définition de fonctions dans le cadre d'un développement avec MATLAB ou SCILAB. Il existe une autre procédure de définition de fonction, directement utilisable en *mode terminal*, et qui met en œuvre la fonction `deff`.

La syntaxe de cette fonction est la suivante :

```
deff(en-tête , corps)
```

La fonction attend deux paramètres. Le premier est une chaîne de caractères, qui définit l'en-tête de la fonction. Cet en-tête obéit à la même syntaxe que les *sci-files*, hormis que le mot-clef `function` ne doit pas être utilisé. Le second paramètre représente le corps de la fonction. Il est fourni sous la forme d'un tableau de chaînes de caractères, dans lequel chaque chaîne représente une des lignes de la fonction. Le tableau peut être fourni sous la forme d'une variable, ou encore d'une suite, placée entre crochets, de chaînes de caractères séparées par des virgules. Ex :

```
-->c=['a=4','b=3','c=5','r=a+b*c']
 c =  !a=4  b=3  c=5  r=a+b*c  !
-->e='[r]=toto()'
 e =  [r]=toto()
-->deff(e,c)
-->toto()
 ans =  19.
```

Dans le cas simple où le corps de la fonction ne comporte qu'une ligne, le second argument peut être remplacé par la chaîne de caractères contenant cette ligne :

```
-->deff('z=demi(x)', 'z=x/2')
-->demi(12.5)
 ans =  6.25
```

2.4.3 Instructions spécifiques

Certaines instructions du langage sont spécifiques aux fonctions. C'est le cas de l'instruction `return`, qui permet d'interrompre le déroulement d'une fonction, et de revenir au programme ayant fait appel à cette fonction. Si la fonction a été déclarée comme fournissant un résultat, la (ou les variables) constituant ce résultat doivent avoir reçu préalablement une valeur. Par exemple, la fonction suivante fournit l'inverse de son paramètre, après avoir testé que celui-ci n'est pas nul ; dans ce dernier cas, elle rend la valeur *infini* :

```
function z = inv(x)
// division
if x == 0 then
   z = %inf
   return
end
z=1/x
```

Note Le mode de fonctionnement de la division est modifiable par la fonction `ieee` ; les trois valeurs du paramètre, 0, 1 et 2 permettent d'obtenir différents modes de fonctionnement :

```
-->ieee(0)
-->1/0
    !--error 27
division by zero...
-->ieee(1)
-->1/0
Warning :division by zero...
 ans = Inf
-->ieee(2)
-->1/0
 ans =  Inf
```

2.4.4 Autres caractéristiques des fonctions

Variables On notera (en première approximation ; nous reviendrons sur ce point au § 4.6.1, page 81), que toutes les variables utilisées dans une fonction sont implicitement locales. Il n'y a pas de déclaration à effectuer. En contrepartie, il n'est pas possible de modifier depuis une fonction une variable de l'espace de travail.

Impression implicite Le système n'effectue pas, pour les instructions ne se terminant pas par un point-virgule, d'impression implicite du résultat de l'expression. On peut donc considérer que les divers terminateurs d'expression (virgule, passage à la ligne, ou point-virgule) sont équivalents à l'intérieur d'une fonction.

Récursivité Le fonctions que permet de définir le langage peuvent bien entendu faire appel à d'autres fonctions, y compris à elles-mêmes. On parle dans ce dernier cas de *fonction récursive*. Un exemple traditionnel de fonction récursive est la fonction factorielle lorsqu'elle est définie sous la forme suivante :

$$\begin{cases} fact(0) = 1 \\ fact(n) = n \times fact(n-1) \end{cases}$$

Ceci peut se traduire en SCILAB par le programme suivant :

```
function r=fact(n)
if n==0 then
    r=1 ;
else
    r= n * fact(n-1) ;
end
```

dont voici quelques utilisations :

```
-->fact(3)
 ans = 6.
-->fact(10)
 ans = 3628800.
```

2.5 Autres éléments du langage

2.5.1 Notes sur la syntaxe

2.5.1.1 Commentaires

Un langage de programmation ne serait pas complet s'il ne permettait pas d'écrire des *commentaires*. Ceux-ci débutent par les caractères `//` et se poursuivent jusqu'à la fin de la ligne. Un commentaire équivaut syntaxiquement à un passage à la ligne, et termine donc l'instruction à la suite de laquelle il est placé.

On peut, sans scrupules, mettre des commentaires dans les fichiers définissant des fonctions, *à condition que ces commentaires apparaissent* ***après*** *la ligne définissant l'en-tête de la fonction.*

Note On notera qu'en Matlab, les commentaires débutent par le caractère **%**.

2.6 Notes sur la mise au point des programmes

Le passage à des problèmes qui ne sont plus entièrement triviaux peut donner naissance à des difficultés de programmation non triviales. Voici quelques points de repères (d'autre suivront, c.f. § 3.6, page 66 et 6.8, page 111) pour la mise au point des programmes.

2.6.1 Erreurs

En cas d'erreur (on l'a constaté à plusieurs reprises), SCILAB imprime un message comportant un libellé d'erreur. Lorsque l'erreur se produit à l'intérieur d'une fonction définie, SCILAB indique également le numéro de la ligne courante et le nom de la fonction, etc. Ex :

```
-->demi('hello')
  !--error 4
undefined variable : %c_r_s
at line 2 of function demi called by :
demi('hello')
```

La fonction `demi` essaye de diviser son paramètre par 2 ; celui-ci étant de type «chaîne de caractères», et la division d'une chaîne de caractères par un nombre n'étant pas définie, le système tente d'appeler une opération définie qui effectuerait ce travail. Le nom de cette fonction, si elle existait, serait `%c_r_s`, ***c*** indiquant que l'opérande gauche est une chaîne de caractères, ***r*** que l'opération concernée est une division, et ***s*** que l'opérande droit est numérique (c.f. le `help overloading` pour plus de précision).

Les éléments d'information fournis sont en général suffisants pour déterminer la position et la nature de l'erreur.

2.6.2 Interruption d'un programme

Dans certains cas, un algorithme peut nécessiter un certain temps pour son exécution (c.f. le chapitre 3), ce qui est normal ; dans d'autres cas, il peut «*boucler*», c'est à dire exécuter une séquence répétitive qui ne s'interrompra jamais car la condition d'arrêt de la séquence ne sera jamais satisfaite (on parle parfois de «*boucle infinie*»).

Prenons un exemple concret. Le programme suivant se propose de calculer la partie entière de son paramètre, par itérations successives.

```
function k=bug1(N)
// recherche de int(N) par encadrement
in=0 // valeur inférieure initiale
su=1E300 // valeur supérieure initiale
while in <> su do
  // moyenne de in et su
  k = int(in+su/2) ;
  if k == N then
    return ; // trouvé
```

```
    end ;
    if k>N then
      su=k ;
    else
      in=k ;
    end
  end
```

L'exécution nous donne le résultat suivant :

```
-->getf('cours/f/bug1.sci')
-->bug1(55)
```

A l'appel de la fonction `bug1(55)`, le système s'abstrait dans une longue méditation, dont on finit au bout de quelques minutes par se demander si elle est bien normale. Choisissons le menu CONTROL ▷ STOP[6]. Le système répond :

```
-1->
```

Le `1` qui apparaît dans le prompt indique qu'il y a eu interruption d'un processus (nous dirons que le programme en cours d'interruption a été *suspendu*). Il est maintenant possible d'examiner le contexte courant, c'est-à-dire les variables locales à la fonction :

```
-1->in
 in =  47.
-1->su
 su =  94.
-1->k
 k =  94.
```

L'exécution peut être relancée par CONTROL ▷ RESUME. Il ne se passe pas grand chose de neuf. Une nouvelle interruption du processus, suivi d'un examen des variables, nous montre que celles-ci ont encore les mêmes valeurs. Que se passe-t-il ?

L'illumination vient brusquement en consultant la ligne 7 de la fonction, en réalisant que l'expression :

```
k = int(in+su/2) ;
```

devrait être :

```
k = int((in+su)/2) ;
```

C'est cette erreur qui faisait que les valeurs des diverses variables n'évoluaient plus. Il faut modifier le programme dans le fichier source, et recharger celui-ci. Mais auparavant, nous allons sortir du programme interrompu, par CONTROL ▷ ABORT.

Remarque : cette correction suffira-t-elle à faire fonctionner le programme ? Vous le saurez en continuant vous-même la mise au point de cette fonction...

[6] Il est aussi possible de taper Ctrl-c.

2.7 Exercices

2.7.1 Éléments de syntaxe

Voici quelques lignes extraites de différents programmes. Dans quel cas pouvez-vous affirmer (sans connaître le reste des programmes) qu'il y a une erreur de syntaxe ?

```
k=3 end ;
end k=3 ;
end end ;
else ; end
```

2.7.2 Quelques calculs

– Calculez les expressions suivantes (le résultat attendu pour les premiers calculs est affiché entre parenthèses) :

$$3 + \frac{8}{5} - (7 \times 4) \qquad \text{(-23.4)}$$
$$\frac{3^2-8}{2^2} + \frac{3^2}{5} - \frac{6\times 11^2}{7^3} \qquad \text{(-0.0666181)}$$
$$e^{\ln(5)} \times (1 - 0.12^2)^{\frac{1}{2}} \qquad \text{(4.9638695)}$$
$$\cosh^2 1.5 - \sinh^2 1.5$$
$$\sin(\tfrac{\pi}{6} + \arccos 0.5)$$
$$\cos^2 4.770796326794897 + \sin^2 11.05398163397448$$

– Calculez $e^x = \sum_{i=0}^{n} \frac{x^i}{i!}$ pour une valeur donnée de x, par exemple sous la forme d'une fonction prenant x et n comme paramètres. Il peut être plus pratique d'utiliser la forme équivalente $e^x = 1 + \sum_{i=1}^{n} \frac{x^i}{i!}$. Comparez, pour différents x et n, les résultats obtenus avec ceux de `exp(x)`.

2.7.3 Nombres premiers

Écrire une fonction `premier(N)` qui détermine si son paramètre est un nombre premier. Discuter de son efficacité.

2.7.4 Nombres parfaits

Un nombre est dit parfait s'il est égal à la somme de ses diviseurs (inférieurs à lui-même). Ainsi, le nombre `6` est parfait, car il est égal à la somme de ses diviseurs `1`, `2`, et `3`.

Écrire une fonction à un paramètre, qui détermine si celui-ci est parfait :

```
-->parfait(4)
 ans = F
-->parfait(6)
 ans = T
-->parfait(8)
 ans = F
```

```
-->for k=1 :50 do
-->  if parfait(k) then
-->    disp(k)
-->  end
-->end
   1.
   6.
   28.
```

Quels sont les nombres parfaits compris entre `1` et `10000` ?

2.7.5 PGCD

On se propose de définir une fonction calculant le *pgcd* de deux nombres entiers. On utilisera l'algorithme d'Euclide qui consiste à soustraire le plus petit nombre du plus grand tant qu'ils sont différents. Programmer cette première version de l'algorithme.

On souhaite maintenant *blinder*[7] le programme, en lui faisant tester certains cas particuliers : si l'un des nombres est négatif, on utilisera sa valeur absolue. Si l'un des nombres est nul, on décidera que le résultat est égal à l'autre nombre. Effectuer et tester ces modifications

Enfin, on souhaite imprimer un message d'erreur si l'un des arguments n'est pas un nombre entier. Réaliser cette dernière transformation.

[7] C'est à dire le protéger contre les cas qu'il ne sait pas traiter correctement.

Chapitre 3

Quelques algorithmes

Le chapitre précédent a présenté les principales constructions du langage. Nous introduisons ici une première structure de données, le *vecteur*, et, après une brève introduction aux notions d'algorithmes et de complexité (éléments que l'on trouvera plus longuement développés dans des ouvrages comme [9], [3], [6] ou encore [4]), ce troisième cours sera l'occasion de mettre en application, d'un point de vue pratique, les concepts acquis aux séances précédentes.

3.1 Une structure de données : le vecteur

Il est difficile de présenter (et donc de résoudre) certaines classes de problèmes lorsqu'on ne dispose pas de structures de données permettant de gérer des ensembles d'éléments. Dans la plupart des langages de programmation, le type *tableau à une dimension* joue ce rôle. En SCILAB, ces tableaux sont désignés sous le terme générique de *vecteur*.

3.1.1 Les tableaux monodimensionnels

Les *tableaux monodimensionnels*, ou encore *tableaux à une dimension* ou à *un indice*, que nous désignerons par la suite sous le terme de *vecteurs*, sont des assemblages compacts d'éléments homogènes. Dans une telle structure, les éléments sont désignés par un numéro, dit *indice*. Les indices sont des entiers naturels, allant de *1* à N si le vecteur comporte N éléments (parfois de *0* à *N-1*, dans certains langages, tel C). Les vecteurs ont en général les propriétés suivantes :

- dans les langages compilés (ADA, C, Fortran, Pascal...), ils ont habituellement une taille définie lors de la compilation et doivent être explicitement déclarés ;
- dans certains langages (APL, C, Lisp, SCILAB), ils peuvent être créés dynamiquement. Cette création est coûteuse, et son coût est proportionnel à la taille du vecteur.
- dans tous les cas, les éléments d'un vecteur peuvent être individuellement consultés ou modifiés, avec un coût fixe (quel que soit l'indice du vecteur) et raisonnable (c'est-à-dire comparable à la consultation ou à la modification d'une variable scalaire).

3.1.2 Vecteurs en SCILAB

Un vecteur SCILAB a les caractéristiques décrites ci-dessus. Il est créé dynamiquement (et cette création a, en effet, un certain coût), ses éléments sont désignés par des indices, qui sont

des entiers positifs compris entre *1* et *N*.

3.1.2.1 Création

Il existe de multiples opérations de création de vecteur. Nous en citerons trois, que nous utiliserons en fonction des besoins :

Vecteur définis par la liste de ses éléments : nous placerons cette liste d'éléments, séparés par des virgules, entre crochets. Exemple :

```
-->v=[2,3,5,9]
 v =
! 2. 3. 5. 9. !
```

Vecteur dont les éléments constituent une progression arithmétique : nous utiliserons la notation *m :n* ou *m :p :n*, déjà vue au paragraphe 2.3.7.1, pour la boucle *for*.

```
-->3 :8
 ans =! 3. 4. 5. 6. 7. 8. !
-->12 :-3 :-4
 ans =! 12. 9. 6. 3. 0. - 3. !
```

Vecteur allant servir de container pour *N* éléments : nous utiliserons la fonction de création ad hoc suivante[1], `vec()`, que nous placerons dans notre fichier de configuration `.scilab` (c.f. chapitre A.1, page 117) :

```
deff('r=vec(x)','r=zeros(1,x)')
-->v=vec(7)
 v =
! 0. 0. 0. 0. 0. 0. 0. !
```

Le paramètre de cette fonction est un entier positif ou nul représentant la taille du vecteur à créer, le résultat étant un vecteur de la taille demandée, dont les éléments sont initialement à zéro.

Vecteur vide : les trois notations décrites ci-dessus permettent le créer le *vecteur vide*, qui s'imprime `[]`, en écrivant `[]`, par une génération telle que `3 :1` («les entiers à partir de *3*, par pas de *1*, inférieurs ou égaux à *1* »), ou encore par `vec(0)`.

[1] Cette fonction fait simplement appel à l'opération primitive, `zeros`, qui permet de créer une matrice dont tous les éléments sont à zéro. Nous nous en servons ici pour créer des vecteurs, qui sont des matrices à 1 ligne.

Note sur l'impression des vecteurs SCILAB imprime les composantes d'un vecteur sous la forme des nombres, séparés par des blancs, encadrés par des points d'exclamation. Un vecteur comportant un nombre suffisamment grand d'éléments s'imprimera sur plusieurs lignes, chaque ligne étant précédée d'un message indiquant les numéros des éléments imprimés sur la ligne :

```
-->1 :16
 ans =
           column 1 to 11
!   1.   2.   3.   4.   5.   6.   7.   8.   9.  10.  11. !
           column 12 to 16
!  12.  13.  14.  15.  16.   !
```

Dans certains cas, un vecteur de grande taille peut nécessiter un grand nombre de lignes pour s'imprimer. SCILAB imprime alors une «page virtuelle» (c'est-à-dire le nombre de lignes qui peut tenir sur l'écran ou sur la fenêtre d'interaction), puis demande si l'impression doit se poursuivre :

```
-->1 :320
 ans =
           column 1 to 11
!   1.   2.   3.   4.   5.   6.   7.   8.   9.  10.  11.   !
           column 12 to 21
!  12.  13.  14.  15.  16.  17.  18.  19.  20.  21. !
           column 22 to 31
..... etc .....
           column 92 to 100
!  92.  93.  94.  95.  96.  97.  98.  99.  100. !
[More (y or n )?]
```

Répondre «n» interrompt l'impression, toute autre réponse fait continuer celle-ci jusqu'à la fin de la «page virtuelle» suivante.

3.1.2.2 Accès aux éléments

Il existe plusieurs méthodes d'accès aux éléments. Nous ferons dans ce chapitre appel à *l'indexation*, qui permet de désigner l'élément d'indice `I` du vecteur `V` par la formule `V(I)`, en consultation comme en modification :

```
-->v(3)
 ans =  0.
-->v(5)=155 ;
-->v(7)=212 ;
-->v
 v =! 0. 0. 0. 0. 155. 0. 212. !
```

L'expression placée entre les parenthèses peut être quelconque, à condition qu'elle fournisse un résultat entier compris entre `1` et la taille du vecteur.

3.1.2.3 Autres fonctions utiles

La fonction `length` fournit le nombre d'éléments d'un vecteur :

```
-->length([1,2,3,5,8,12])
 ans =  6.
-->length(3 :2 :197)
 ans =  98.
```

La pseudo-variable `$`, utilisée comme indice d'un vecteur, désigne le dernier élément de ce vecteur. l'écriture `V($)` est donc équivalente à `V(length(V))`.

La fonction `size` s'applique à une matrice, et fournit les dimensions de son paramètre sous la forme d'un vecteur à deux éléments, le nombre de lignes et le nombre de colonnes.

```
-->size([1,2,3,5,8,12])
 ans = ! 1.  6. !
```

Le résultat nous rappelle que SCILAB considère les vecteurs comme des matrices à une ligne. Par ailleurs, la fonction `length()` exhibe un comportement particulier lorsqu'elle s'applique à des tableaux de caractères. On utilisera de préférence la fonction `size()`, à travers la fonction `count()` suivante[2], qui fournit le nombre d'éléments de son paramètre, quel que soit celui-ci :

```
deff('r=count(s)','r=prod(size(s))')
```

Enfin, la fonction `isempty()`, appliquée à un tableau, nous indique si celui-ci est vide, ou s'il contient des éléments :

```
-->isempty(1 :10)
 ans = F
-->isempty([])
 ans = T
-->isempty(2 :-1 :8)
 ans = T
```

3.2 Quelques exemples de programmes

3.2.1 Somme des éléments d'un vecteur

On se propose d'écrire une fonction qui calcule la somme des éléments d'un vecteur. On effectue pour ce faire une boucle décrivant les éléments du vecteur, en ajoutant chaque élément à un totalisateur, initialement à 0. Voici (à placer dans un fichier) une première version de la chose :

```
function r=sum1(V)
r=0
for k=1 :count(V) do
```

[2]Dans cette écriture, la fonction `prod()`, que nous verrons très bientôt, calcule le produit des éléments du vecteur qui lui est transmis en paramètre.

```
  r=r+V(k)
end
```

La boucle nous permet de *décrire* tous les éléments du vecteur (on parle parfois de «*balayer*» le vecteur), l'élément courant étant consulté à travers son indice `k`, par l'expression `V(k)`.

Quelques exemples :

```
-->sum1(3)
 ans = 3.
-->sum1([2,5,3,8])
 ans = 18.
-->sum1(1 :1000)
 ans = 500500.
-->sum1([])
 ans = 0.
```

Notons que la fonction prédéfinie `sum()` effectue le même travail, de manière plus performante.

Exercice Programmez de même la fonction `prod1` qui calcule le produit des éléments d'un vecteur. Peut-on, dans certains cas, accélérer le traitement ?

On notera que, comme dans le cas de la fonction `sum()`, il existe une fonction prédéfinie `prod()` qui fournit ce service particulier.

3.2.2 Réversion d'un vecteur

On se propose d'écrire un programme qui nous fournisse le miroir d'un vecteur, c'est-à-dire que le premier élément du résultat est le dernier élément du vecteur initial, etc. Voici un exemple de ce que nous désirons obtenir :

```
-->v = [2,5,3,8,9,0] ;
-->reverse(v)
 ans =! 0.  9.  8.  3.  5.  2. !
```

Nous avons programmé la fonction suivante. Fait-elle bien ce que nous désirons ?

```
function Z=rev1(V)
// Miroir d'un vecteur
Z=V
N=count(Z)
for k=1 :N do
  l=N+1-k
  tmp=Z(k) ; Z(k)=Z(l) ; Z(l)=tmp ;
end
```

Proposez une correction de ce programme.

Voici une autre version de la réversion d'un vecteur :

```
function Z=rev2(V)
// Miroir d'un vecteur
N=count(V) ;
Z=V for l=N-1 :-1 :1 do
    for k=1 :l do
        tmp=Z(k) ; Z(k)=Z(k+1) ; Z(k+1)=tmp ;
    end
end
```

Cette fonction réaliste-t-elle vraiment la réversion d'un vecteur ?

3.3 Algorithmes et complexité

3.3.1 Introduction

Lorsque nous comparons les durées d'exécution des programmes `rev1()` (une fois mis au point !) et `rev2()` ci-dessus, en particulier pour des vecteurs de taille relativement importante, il est clair que ces deux programmes ne se comportent pas de la même manière : l'un devient beaucoup plus lent que l'autre. Si l'on analyse plus en détail leurs fonctionnement, on découvre que `rev1()` va manipuler quelque chose comme $k \times n$ éléments, alors que `rev2()` en manipule $k' \times \frac{n \times (n+1)}{2}$, k et k' étant des constantes liées à la programmation. Le programme `rev1()`, à la lumière de cette réflexion apparaît donc *meilleur* que le programme `rev2()`. Peut-on systématiser cette réflexion, afin, par exemple, de prévoir le temps d'exécution des programmes, et l'efficacité de leur implantation ? Oui (bien sûr !), c'est l'objet de *l'analyse de la complexité des programmes*, un aspect d'un domaine particulier de l'informatique qui a pour nom *l'algorithmique*.

3.3.2 Algorithmique

En ce qui nous concerne, nous entendrons par *algorithme* un *processus de calcul fini, qui conduit à la solution d'un problème donné*. On s'intéresse naturellement à la classe des algorithmes qui vont pouvoir être représentés par des programmes susceptibles de s'exécuter sur les machines dont nous disposons. Une illustration en est le célèbre algorithme d'Euclide.

Idéalement, l'analyse d'un algorithme comprend deux étapes :

- la démonstration de sa validité ;
- l'étude de sa complexité.

La *validité* d'un algorithme est difficile à établir formellement, et tous les tests que l'on peut faire sont, au mieux, susceptibles de démontrer qu'un algorithme (ou l'implantation d'un algorithme) ne fonctionne pas. Avec quelques précautions, la *complexité* d'un algorithme est assez souvent facile à établir. *Lorsque plusieurs algorithmes de complexités différentes sont envisageables pour la résolution d'un problème, la complexité est presque toujours un critère de choix déterminant dans l'implantation.*

Identification	Description
$O(1)$	Temps d'exécution indépendant du nombre d'éléments
$O(\ln n)$	Temps d'exécution proportionnel au logarithme du nombre d'éléments
$O(n)$	Temps d'exécution proportionnel au nombre d'éléments
$O(n \ln n)$	Temps d'exécution proportionnel à $n \times \ln n$
$O(n^2)$	Temps d'exécution proportionnel au carré du nombre d'éléments
$O(n^3)$	Temps d'exécution proportionnel au cube du nombre d'éléments
$O(2^n)$	Temps d'exécution exponentiel

FIG. 3.1 – Table des complexités des algorithmes

3.3.3 Complexité

La *complexité* d'un algorithme est liée à son temps de calcul, à son encombrement mémoire, etc. Il existe habituellement dans tout algorithme un ou plusieurs paramètres qui vont influer directement sur ce temps de calcul ou cet encombrement mémoire. Lorsqu'il s'agit par exemple de calculer la somme d'un ensemble d'éléments, le nombre d'éléments de cet ensemble va clairement influer sur le temps nécessaire pour effectuer ce calcul.

La notation suivante, dite notation en *grand O*, introduite par D.E. Knuth ([13]), définit des classes de fonctions :

$$O(g) = \{f : \aleph \to \aleph \,/\, c > 0,\ \exists n_0 \in \aleph,\ \forall n \geq n_0,\ f(n) \leq c \times g(n)\}$$

Dire qu'une fonction f appartient à la classe $O(g)$, notation dans laquelle g représente une fonction au comportement connu, revient à dire que le comportement de cette fonction f est borné, à un facteur près, par celui de g.

Ainsi, $f(n) \to n \in O(n^2)$, $f(n) \to 431 \times n^2 \in O(n^2)$, mais $f(n) \to n \notin O(\ln n)$. On notera que la première écriture est exacte, mais qu'il est plus précis d'écrire : $f(n) \to n \in O(n)$.

En algorithmique, nous nous intéressons essentiellement à des algorithmes dont le comportement dépend de la valeur ou du nombre d'éléments n de l'une des entrées. Si le temps d'exécution d'un algorithme est proportionnel à cette valeur ou à ce nombre d'éléments des données en entrée, nous dirons que l'algorithme est en $O(n)$. On pourra dès lors affirmer, par exemple, que si la taille des données à traiter double, la durée de ce traitement sera également doublée.

Dans la pratique, nous nous intéresserons aux classes de complexités décrites dans le tableau 3.1, page 55. La complexité augmentant, et l'algorithme devenant donc «plus mauvais», au fur et à mesure que l'on descend dans le tableau.

Un peu de terminologie : on parlera d'algorithme *logarithmique* pour désigner un algorithme dont la complexité est en $O(\ln n)$, d'algorithmes *linéaires*, *quadratiques* ou *exponentiels* pour ceux dont la complexité est en $O(n)$, $O(n^2)$ et $O(2^n)$ respectivement.

3.3.4 Évaluation des complexités

Un algorithme se décompose souvent en parties, juxtaposées ou imbriquées, dont il est possible de calculer séparément les complexités. On peut établir aisément les résultats suivants,

$O(c \times f) = O(f)$
$O(f + g) = O(f) + O(g)$
$O(f \times g) = O(f) \times O(g)$

FIG. 3.2 – Opérations sur les complexités

Algorithme	**10**	**20**	**30**	**50**	**80**	**100**
$O(n)$	0.0001	0.0002	0.0003	0.0005	0.0008	0.001
$O(n \ln n)$	0.00023	0.00060	0.00102	0.00196	0.00351	0.00461
$O(n^2)$	0.001	0.004	0.009	0.025	0.064	0.1
$O(n^3)$	0.01	0.08	0.27	1.25	5.12	10
$O(n^5)$	1	32	243	52 m	9 h	27 h
$O(2^n)$	0.01024	10.4858	178 m	348 ans	4E+09 siècles	4E+15 siècles
$O(3^n)$	0.59049	9 h	66 ans	2E+09 siècles	5E+23 siècles	2E+33 siècles

FIG. 3.3 – Durée d'exécution de certains algorithmes

pour deux fonctions f et g, et une constante c, résumés dans la table 3.2, page 56. Enfin, il convient en général dans une somme de négliger la plus petite des complexités ; nous dirons qu'un algorithme est en $O(n^2)$ si le calcul nous apprend qu'il est en $O(n) + O(n^2)$. Pour nous donner une idée de l'influence de la complexité sur le temps d'un algorithme, le tableau 3.3, page 56, nous donne les durées d'exécution de divers algorithmes, pour diverses tailles des données (de 10 à 100 éléments), la durée d'une opération élémentaire étant la même dans tous les cas (10^{-5} secondes). Ces durées sont exprimées en secondes, sauf indication contraire (m pour minutes, h pour heure, *ans* et *siècles* si nécessaire). On notera que certaines classes d'algorithmes deviennent vite inutilisables (c'est un euphémisme) lorsque la taille des données augmente.

3.3.5 Complexité des opérations usuelles

Toutes les instructions d'une machine (et, a fortiori, toutes les instructions d'un langage de programmation de haut niveau) ne nécessitent pas la même durée pour s'exécuter. Une élévation à la puissance, par exemple, est «plus lente» qu'une addition. Les différences entre les temps d'exécution individuels peuvent être considérables. Nous considérerons cependant que toutes ces opérations opèrent en un temps borné, indépendant des valeurs qui entrent en jeu dans l'opération. C'est ainsi que toutes les instructions suivantes :

```
2+3
a*x^2+b*x+c
sqrt(%eps)*(2-3*abs(p))
if a>b then a=a-b ; else b=a+b/2 ; end
```

sont considérées comme des opérations en $O(1)$, sous réserve naturellement que toutes les variables intervenant dans les expressions soient des scalaires.

En revanche, chaque instruction répétitive devra être analysée avec soin, pour déterminer sa complexité. Ainsi :

```
k=1 ; while k<N do k=k*2 ; end
s=0 ; for k=1 :N do s=s+k ; end
s=0 ; for k=1 :N do for l=1 :N do s=s+1/(k+l-1) ; end ; end
```

les trois boucles ci-dessus sont respectivement en $O(\log n)$, $O(n)$ et $O(n^2)$ respectivement. La complexité du programme proposé au § 3.2.1, qui décrit chaque élément du vecteur, est clairement $O(n)$. De même, les deux opérations de réversions de vecteurs du § 3.2.2 s'avèrent-elles en $O(n)$ et $O(n^2)$.

3.3.6 Mesures expérimentales

Il est intéressant de conforter ces résultats théoriques par quelques mesures expérimentales. La fonction `chrono`, dont le code source est fourni en annexe (page 118) permet de mesurer le temps d'exécution d'une expression. Il est possible d'évaluer de cette manière l'ordre de grandeur d'un algorithme, en lui fournissant des données de tailles différentes, et en traçant la courbe des résultats obtenus. Quelques exemples d'utilisation :

```
-->chrono('2+3 ;')
 0.0204086 ms.
-->chrono('exp(2)+log(3) ;')
 0.0439072 ms.
```

Les mesures effectuées au moyen de cette fonction sont peu précises dans les cas d'expressions simples comme celles-ci, qui utilisent peu de temps de calcul. L'exercice 3.7.2 donne quelques exemples de mesures obtenues en utilisant des algorithmes plus complexes.

3.4 Quelques analyses de programmes

3.4.1 Rotation d'un vecteur

On se propose d'écrire un programme qui réalise la rotation d'un vecteur, c'est à dire une permutation circulaire de ses éléments. Voici un exemple de ce que nous désirons obtenir :

```
-->v
v =! 1. 2. 3. 4. 5. 6. 7. 8. 9. 10. !
-->rotate1(v,3)
ans =! 4. 5. 6. 7. 8. 9. 10. 1. 2. 3. !
-->rotate1(v,7)
ans =! 8. 9. 10. 1. 2. 3. 4. 5. 6. 7. !
```

Voici le texte source de la fonction :

```
function R=rotate1(V,d)
R=V
n=count(V)
for k=1 :n
```

```
    l=1+modulo(k+d-1, n)
    R(k)=V(l)
end
```

La fonction se comporte-t-elle correctement si le décalage `d` est négatif ou supérieur à la taille du vecteur ? Que faut-il modifier dans la fonction ?

Complexité L'algorithme est clairement linéaire, puisqu'il comporte une boucle unique réalisant à chaque itération un transfert d'éléments.

Plus difficile La fonction nécessite deux vecteurs pour fonctionner : le vecteur *source*, ici `V`, et le vecteur *destination*, ici `R`. Peut-on trouver un algorithme qui soit toujours en $O(n)$, mais puisse travailler avec un seul vecteur ? (au moins deux solutions, très différentes, sont possibles).

3.4.2 Recherches dans un vecteur

Un problème traditionnel consiste à rechercher un élément dans un vecteur. La solution la plus simple consiste à «balayer» le vecteur, du premier au dernier élément, en comparant l'élément courant à l'élément recherché. La fonction suivante implémente un tel algorithme :

```
function B=memb1(E,V)
// Recherche de E dans V
N=length(V)
n=1
while n <= N do
  if E == V(n) then
    B=%T ; return
  end
  n=n+1
end
B=%F
```

Testons l'objet :

```
-->v=[2,9,8,1,12,39,23,7,0,11] ;
-->memb1(7,v)
 ans =  T
-->memb1(6,v)
 ans =  F
```

La fonction semble se comporter correctement. Quel est sa complexité ? Lorsque le nombre cherché n'existe pas dans le vecteur, il est nécessaire de balayer tout celui-ci pour s'en rendre compte. Si le nombre existe dans le vecteur, il peut être situé n'importe où dans celui-ci, et il faut donc, en moyenne, balayer la moitié du vecteur pour le trouver. Dans tous les cas, la durée de la recherche est proportionnelle au nombre d'éléments. L'algorithme utilisé est donc en $O(n)$.

Pour ce problème précis, il n'est guère possible de faire mieux, sauf à changer la nature du problème.

On peut imaginer que le vecteur où l'on recherche les éléments soit déjà trié. Il devient dès lors possible de proposer un meilleur algorithme, dit de *recherche binaire*, ou *binary search*.

Celui-ci consiste à éliminer, à chaque étape, l'une des moitiés du vecteur. Une comparaison de l'élément cherché avec l'élément médian permet de savoir si l'élément cherché est plus grand ou plus petit que celui-ci. Il est donc possible d'éliminer de la recherche l'une des moitiés du vecteur. La partie du vecteur où peut se situer l'élément cherché devient successivement de taille $\frac{n}{2}$, puis $\frac{n}{4}$, $\frac{n}{8}$, etc., jusqu'à ne plus comporter qu'un élément, qui peut, ou non, être l'élément recherché. L'algorithme ne nécessite donc que $\ln n$ étapes. Sa complexité est $O(\ln n)$, donc meilleure que celle de l'algorithme précédent. À peine plus complexe à programmer que le précédent, il fournira pour de grands vecteurs de meilleurs performances. En particulier, si le vecteur où sont rangés les éléments parmi lesquels on effectuera la recherche est défini une fois pour toutes au début de l'exécution, et si l'on va y effectuer un nombre important de recherches, il peut être rentable de trier ce vecteur, pour pouvoir appliquer ce second algorithme.

3.4.2.1 Description de l'algorithme de recherche binaire

Nous voulons déterminer si un tableau d'éléments $X_{i,i=1,N}$ avec $N \geq 0$, tel que $X_1 \leq X_2 \leq X_3 \cdots \leq X_N$ contient un élément T ; le résultat sera la valeur logique *vrai* si X contient l'élément T, *faux* sinon. L'algorithme est le suivant :

- l'élément est recherché dans le sous-ensemble du tableau X dont les indices sont compris entre *imin* et *imax*, bornes inclues.
- les valeurs initiales de *imin* et *imax* sont respectivement *1* et *count(X)*, nombre d'éléments du tableau.
- dans le sous-ensemble de recherche, la valeur médiane, d'indice k, est consultée. k est calculée comme $\left\lfloor \frac{imin+imax}{2} \right\rfloor$.
- si $T \equiv X_k$, l'algorithme s'arrête et rend *vrai*.
- si $T > X_k$, *imin* devient $k+1$, sinon *imax* devient $k-1$.
- l'algorithme s'arrête si $imin > imax$, et rend *faux*.

On tentera de montrer qu'à chaque étape de cet algorithme, la taille de l'intervalle de recherche est divisée par 2, et que l'algorithme s'arrête obligatoirement ; que si le nombre existe dans le tableau, il sera trouvé, et que s'il n'existe pas, l'algorithme rendra *faux*.

3.4.2.2 Une solution possible

Il est recommandé de programmer l'algorithme décrit ci-dessus, et de le tester pour différents cas, avant de consulter la solution proposée ici.

Le programme ci-dessous est une implantation possible de l'algorithme :

```
function [r]=memb2(E,V)
// Recherche dans un vecteur trié par valeurs croissantes
n=1 ;
p=length(V) ;
while n<=p do
```

```
    k=int((n+p)/2)
    if E==V(k) then
      r=%T ; return
    end
    if E < V(k) then
      p=k-1
    else
      n=k+1
    end
  end
  r=%F
```

Notes concernant la recherche binaire La recherche binaire est relativement délicate à programmer (sauf si l'on dispose de l'excellent algorithme proposé :-). Knuth ([12]) signale que, bien que l'algorithme ait été décrit dès 1946, il n'y a pas eu de version correcte de cet algorithme publiée avant 1962, et Jon Bentley ([4]) précise que, sans l'aide d'une machine (pour la mise au point...), la plupart des programmeurs sont incapables d'en écrire sur le papier une version correcte.

3.4.3 Tris de vecteurs

Un autre grand problème classique est celui du tri d'un vecteur numérique. Il existe en SCILAB une opération (`sort`) qui résout le problème ; nous l'ignorerons pour l'instant, pour programmer nos propres versions.

3.4.3.1 Vecteurs à nombre limité de valeurs

On se propose de trier un vecteur ne comportant que des 0 et des 1. Le résultat devra comporter tous les 0 en début, et tous les 1 à la fin (il s'agit d'un tri *ascendant*, l'opération consistant à ordonner les éléments du plus grand au plus petit étant, on s'en doute, le tri *descendant*).

Première solution On balaye le vecteur à trier, en échangeant l'élément courant avec un élément de la fin s'il ne convient pas.

```
function Z=bsort1(V)
Z=V
k=1
l=count(Z)
s=0
while k < l do
   if Z(k) <> s then
      tmp = Z(k) ; Z(k) = Z(l) ; Z(l)=tmp ;
```

```
            l = l-1 ;
        else
            k=k+1 ;
        end
    end
```

Dans cette solution, `s` représente l'élément cherché (`0` ; en prenant `s=1`, on obtiendrait le tri inverse), `k` l'indice qui sert à balayer le vecteur, `l` la position de fin de balayage (après laquelle on est sûr que tous les éléments sont différents de `s`).

On peut montrer que l'algorithme se termine : à chaque itération, soit l'élément courant, `Z(k)`, est égal à `s`, et `k` est incrémenté de `1`, soit ce n'est pas le cas, et cet élément est échangé avec `Z(l)`, et `l` est décrémenté de `1`. La différence, `l-k`, initialement égale à N, nombre d'éléments moins `1`, est ainsi décrémentée à chaque exécution de la boucle `while`. L'algorithme se termine donc en N itérations.

On peut montrer également que l'algorithme effectue correctement son travail : à chaque étape, tous les éléments d'indice i du vecteur, pour $i<k$ sont à `0`, tous les éléments d'indice j du vecteur, pour $j>l$, sont égaux à `1` (c'est initialement vrai, de manière triviale : il n'y a pas d'élément avant le premier ni après le dernier item du vecteur). Lorsque k devient égal à l, tous les éléments avant k sont à 0, tous les éléments après k sont à 1. Quant à l'élément d'indice k, il peut lui-même être à `0` ou `1`, la configuration obtenue pour le vecteur correspond dans tous les cas à ce qui était désiré.

La complexité de l'algorithme est elle-même simple à évaluer. L'algorithme consiste en une boucle qui est exécutée N fois. Toutes les opérations exécutées à l'intérieur de cette boucle sont des opérations élémentaires, dont la complexité est en $O(1)$. L'algorithme est donc en $O(n)$.

Seconde solution On balaye le vecteur en comptant le nombre de 0 (ou de 1). On crée un nouveau vecteur, débutant par le bon nombre de 0, suivi du bon nombre de 1.

Extension de l'algorithme Imaginons maintenant que le vecteur comporte des éléments appartenant à un ensemble petit, connu à l'avance. Il s'agit, par exemple, de trier un très grand nombre de valeurs comprises entre 1 et 10. Généraliser l'algorithme précédent, en vérifiant qu'il se comporte bien en $O(n)$.

3.4.3.2 Cas général

Les problèmes de tri ont fait l'objet de nombre d'études. Une lecture indispensable à ce sujet reste le Knuth ([12]), mais le sujet est traité dans tous les ouvrages parlant peu ou prou d'algorithmique et de programmation. Nous nous bornerons ici à présenter quelques solutions typiques.

Tri quadratique L'une des premières idées qui vient à l'esprit est la suivante :

- un *balayage* du vecteur permet de repérer le plus grand (ou le plus petit élément). Au cours de ce balayage, il est possible de permuter deux éléments successifs du vecteur, de telle sorte que le plus grand élément (ou le plus petit) se retrouve à la dernière position du vecteur ;

- en répétant cette opération, sur N-1 éléments (en s'arrêtant donc avant l'élément déposé en dernière position dans le vecteur), on amène le second plus grand (ou plus petit) élément en avant dernière position dans le vecteur ;
- en opérant de la sorte, sur N-2, N-3, etc. éléments, jusqu'à ce que le vecteur considéré ne comporte plus qu'un élément, on obtient ainsi dans le vecteur final les éléments triés du plus petit au plus grand, (ou l'inverse).

L'algorithme que l'on en déduit est sensiblement le suivant :

for k=début :-1 :fin do
  for l=1 :k do
    if Z(l) > Z(l+1) then
      tmp=Z(l) ; Z(l)=Z(l+1) ; Z(l+1)=tmp ;
    end
  end
end

La boucle interne balaye les éléments du vecteur de *1* jusqu'à une certaine limite, *k*, qui décroît au cours de l'algorithme. Le cœur de la boucle comporte simplement un test qui échange deux éléments consécutifs s'ils ne sont pas ordonnés (le tri est ici *ascendant*). Un léger flou subsiste sur les valeurs *début* et *fin*. Lors de la première itération, le programme doit déposer le plus grand élément en position *N* : *l+1* est donc alors égal à *N* ; *k* doit donc avoir la valeur *N-1* ; *début* vaut donc également *N-1*. La dernière itération doit porter sur un vecteur de longueur *2*. *l+1* est donc alors égal à *2*, donc *l* vaut *1* ; *k* a également cette valeur, qui est donc celle de *fin*.

On peut dès lors proposer le programme suivant :

```
function Z=tri0(V)
Z=V
N=count(Z)
for k=N-1 :-1 :1 do
    for l=1 :k do
        if Z(l) > Z(l+1) then
            tmp=Z(l) ; Z(l)=Z(l+1) ; Z(l+1)=tmp ;
        end
    end
end
```

programme dont voici quelques exemples d'exécution :

```
-->tri0([2 7 3 5 1 7 3 5 1 9 8 0 2])
 ans =! 0. 1. 1. 2. 2. 3. 3. 5. 5. 7. 7. 8. 9. !
-->tri0([2 7 5 1])
 ans =! 1. 2. 5. 7. !
-->tri0([2 0])
 ans =! 0. 2. !
-->tri0([0])
```

```
 ans = 0.
-->tri0([])
 ans = []
```

Les derniers tests montrant que le programme marche correctement aux «limites» : vecteur vide, ou ne comportant qu'un ou deux éléments.

Complexité L'algorithme comporte deux boucles imbriquées. La boucle externe est exécutée *N-1* fois ; la boucle interne s'intéresse à *N-1* éléments, puis *N-2*, *N-3*, etc, jusqu'à *1* élément, soit en moyenne *N/2* éléments. Le temps d'exécution est donc proportionnel à $N \times \frac{N}{2}$, soit N^2. L'algorithme est donc en $O(n^2)$.

3.5 Opérations globales sur vecteurs

SCILAB nous propose un certain nombre d'opérations s'appliquant sur des vecteurs. Pour bien assimiler leur fonctionnement, nous les classifierons en diverses catégories dont nous examinerons les rouages internes.

3.5.1 Réduction

Nous avions vu, au § 3.2.1, l'exemple du calcul de la somme des éléments d'un vecteur. On peut, de la même manière, calculer le produit, le maximum, le minimum, etc. À chaque fois, il faut reprogrammer la gestion de la boucle sur les éléments, les contrôles de validité, etc. Or SCILAB permet de passer une fonction comme argument à une autre fonction. On peut donc envisager d'*abstraire* l'opération effectuée sur les éléments du vecteur, et de réaliser un outil qui nous permette d'appliquer une fonction sur l'ensemble des éléments d'un vecteur. Cette opération est parfois dite *réduction*, et l'on parlera de *réduction par plus* pour désigner la *sommation* des éléments d'un vecteur, etc.

Voici une première version de cette opération :

```
function z=reduc1(v,f,init)
z=init
for l=v do
   z=f(z,l)
end
```

Cette fonction prend comme paramètres un vecteur, une fonction, et une valeur initiale (qui dépendra de la fonction). Définissons quelques fonctions auxiliaires : `add()`, addition de deux nombres, `mul()`, multiplication de deux nombres, etc[3] :

```
-->deff('z=add(x,y)', 'z=x+y')
-->deff('z=mul(x,y)', 'z=x*y')
```

[3]La syntaxe du langage ne nous autorise pas à utiliser directement comme paramètre d'une fonction un opérateur du langage (comme `+`, `*`), ou une fonction primitive du système (comme `cos`, `abs`), comme cela est possible dans d'autres langages (*Lisp*, *Scheme*, *Caml*, etc.). Il nous faut «emballer» l'utilisation des opérateurs ou fonctions primitives dans une fonction définie.

Voici maintenant quelques exemples d'application de la réduction.

```
-->v=1 :10
 v =! 1. 2. 3. 4. 5. 6. 7. 8. 9. 10. !
-->reduc1(v,add,0)
 ans = 55.
-->reduc1(v,mul,1)
 ans = 3628800.
```

Il est possible de trouver plus original. La fonction `add1(x,y)` ajoute 1 à son premier paramètre :

```
-->deff('z=add1(x,y)', 'z=x+1')
-->reduc1(v,add1,0)
 ans = 10.
```

Utilisée avec l'opération de réduction, elle nous fournit la *longueur* du vecteur. Plus généralement, la fonction utilisée pour la réduction peut être conçue de manière à réaliser une fonction particulière sur un élément, le réduction réalisant l'extension à tout le vecteur. Quelques exemples :

```
-->// somme des carrés des éléments d'un vecteur
-->deff('z=sqr1(x,y)', 'z=x+y*y')
-->reduc1(v,sqr1,0) ans = 385.
-->// inverse de !10
-->deff('z=pinv1(x,y)', 'z=x/y')
-->reduc1(1 :10,pinv1,1)
 ans = 2.756E-07
-->// suite harmonique
-->deff('z=sinv1(x,y)', 'z=x+1/y')
-->reduc1(1 :10,sinv1,0)
 ans = 2.9289683
```

On voit dans ces derniers exemples que les fonctions appliquées par la *réduction* réalisent en fait deux opérations différentes : réaliser la *transformation* d'un élément, et *combiner* le résultat obtenu avec les autres éléments, typiquement par une opération telle que l'addition ou la multiplication. On peut proposer une version de la réduction qui disjoint ces deux opérations :

```
function z=reduc(v,f,g,init)
z=init
for l=v do
    z=f(z,g(l))
end
```

Cette version utilise deux fonctions : f sert à combiner les résultats, g à transformer un élément. La valeur *init* est en général l'élément neutre de la fonction f. Quelques exemples :

```
-->deff('z=idt(x)', 'z=x') // fonction identité
-->reduc(v,add,idt,0) // somme des éléments
 ans = 55.
-->reduc(v,mul,idt,1) // produit des éléments
 ans = 3628800.
-->deff('z=sqr(x)', 'z=x*x')
-->reduc(v,add,sqr,0) // somme des carrés des éléments
 ans = 385.
-->deff('z=un(x)', 'z=1')
-->reduc([2 7 3 5 1 9 8], add, un, 0) // taille du vecteur
 ans = 7.
-->deff('z=petit(x,y)', 'if x<y then z=x ; else z=y ; end')
-->petit(3,5)
 ans = 3.
-->reduc([2 7 3 5 1 9 8], petit, idt, %inf) // minimum
 ans = 1.
```

La dernière expression nous permet de trouver le plus petit élément d'un vecteur. On utilise une fonction auxiliaire, `petit`, qui fournit le plus petit de ses arguments. L'élément neutre de cette fonction est la valeur `%inf`, $+\infty$. Comment écririez-vous une fonction `petit` qui n'utiliserait pas la construction `if then else`, mais ferait appel uniquement à des fonctions mathématiques ?

Question Comment utiliser `reduc` pour savoir si tous les éléments d'un vecteur booléen sont vrais ? Si tous sont faux ? Si certains d'entre eux sont vrais ? Si certains d'entre eux sont faux ?

3.5.2 Opérations de réduction en Scilab

Ce concept de réduction se retrouve dans nombre de fonctions du langage s'appliquant à un vecteur :

- *somme* des éléments d'un vecteur : c'est l'opération `sum(V)`.
- *produit* des éléments d'un vecteur : `prod(V)`.
- *moyenne* des éléments d'un vecteur : `mean(V)`.
- *maximum* des éléments d'un vecteur : `max(V)`.
- *minimum* des éléments d'un vecteur : `min(V)`.
- *et logique* : `and(V)`. Le résultat est *vrai* si tous les éléments du paramètre sont *vrais*.
- *ou logique* : `or(v)`. Le résultat est *vrai* si au moins l'un des éléments du paramètre est *vrai*.

3.5.3 Complexité des opérations sur vecteurs

On peut comparer les temps d'exécution des fonctions primitives (`sum`, `prod`) et celui des fonctions définies par l'utilisateur (`sum1`, `prod1`). Le tableau suivant donne les valeurs des durées d'exécution (en millisecondes), pour des données de diverses tailles :

Taille des données	400	800	1600	3200
Fonction `sum`	0.0486755	0.0652313	0.0991058	0.1684570
Fonction `sum1`	5.1953125	10.283203	20.605469	40.9375

On peut approximer le temps mis par ces fonctions par :
- sum : 0.0316755 + 0.0000433 * N ;
- sum1 : 0.107422 + 0.0127075 * N.

Dans ces expressions, le terme constant correspond à la «mise en œuvre» de la fonction, auquel il convient d'ajouter un certain coût par élément. Du strict point de vue de la complexité des algorithmes mis en jeu, les deux opérations sont équivalentes, et sont en $O(n)$. On préférera naturellement dans la pratique la fonction `sum`, dont le rapport des performances par rapport à la fonction définie `sum1` est, sur de grosses données, de l'ordre de 250 !

3.6 Notes sur la mise au point des programmes

3.6.1 Visualiser des variables

Il peut être nécessaire dans certains cas de pouvoir observer les valeurs de certaines variables à l'intérieur d'une fonction, ou encore d'interrompre une fonction à un certain endroit, pour observer ce qui se passe.

La fonction `disp()` permet d'afficher une chaîne de caractères. Nous verrons plus loin dans le cours la manipulation des chaînes de caractères, mais on peut à ce stade utiliser deux opérations qui ont déjà été présentées :
- la concaténation de deux chaînes, représentée par l'opération `+` ;
- la transformation d'un objet en chaîne, représenté par l'opération `string()`.

Si l'on veut tracer la valeur de la variable `toto` à un certain emplacement d'un programme, on peut ajouter la ligne :

```
disp('toto = ' + string(toto)) ;
```

3.6.2 Programmer une interruption de l'exécution

Pour suspendre l'exécution d'un programme à une certaine ligne (plutôt que «quand ça se présente», comme on peut le faire avec Ctrl-C ou Control▷Stop), il suffit d'ajouter la ligne :

```
pause
```

dans le programme. Lors de son exécution, le programme s'interrompt à cet endroit, et il est possible de consulter les différentes variables qu'il manipule. L'exécution du programme peut être continuée par la commande `return()` (ou `resume()`) ; elle peut être définitivement arrêtée par `abort()` (ou Control▷Abort).

3.6.3 Modifier les variables du programme interrompu

L'utilisateur aura remarqué, après quelques manipulations, qu'il est possible de modifier les variables d'un programme suspendu, mais que ces modifications semblent n'être pas prises en compte lors de la reprise du programme suspendu : celui-ci continue son exécution avec les

valeurs que les variables avaient au moment de son interruption. En effet, SCILAB a en fait effectué des copies de ces variables, et ce sont ces copies qui ont été éventuellement modifiées par l'utilisateur.

Il est possible cependant de relancer l'exécution d'un programme, après modification de certaines de ses variables. Il convient d'utiliser pour ce faire la commande `resume()` (ou `return()`), avec la syntaxe suivante :

```
[ liste_de_variables ] = resume ( liste_de_valeurs )
```

Par exemple :

```
[p,q,r]=resume(2,q-1,r*2)
```

Cette expression va relancer l'exécution du programme suspendu, les trois variables `p`, `q` et `r` ayant pour nouvelles valeurs les résultats des expressions `2`, `q-1` et `r*2` respectivement. Une illustration :

```
-->i=0 ; s=0 ; while i<10 do i=1+1 ; s=s+i ; end ; 'fini', s
-1->[i]=resume(10)
ans = fini
s = 88472.
```

Dans cet exemple, une erreur de frappe (`i=1+1` au lieu de `i=i+1`) fait que le test d'arrêt de la boucle ne sera jamais satisfait. L'utilisateur interrompt le calcul, et relance l'exécution, en fixant `10` comme nouvelle valeur de `i`. Le test d'arrêt étant dès lors satisfait, le programme se termine (tout en fournissant un résultat sans grand intérêt, il est vrai...).

3.6.4 Affichage des programmes suspendus

Après plusieurs minutes de tests divers, on peut se retrouver perdu, ne plus savoir quelle fonction est interrompue, ou à quelle ligne. La fonction `whereami()` permet de connaître la liste des programmes interrompus, et les numéros de lignes correspondantes. Ex :

```
-->bug1(23.6)
-1->whereami()
whereami called under pause
pause called at line 7 of macro bug1
```

3.7 Exercices

3.7.1 Complexités et puissances des machines

Une légende urbaine fort répandue veut que les problèmes de complexité des algorithmes n'aient plus guère d'importance, puisque «les machines vont de plus en plus vite, c'est finalement le temps du programmeur qui compte, et il vaut mieux choisir un algorithme mauvais, mais simple à programmer, plutôt que de se casser la tête».

Prenons donc un cas concret. On imagine que l'on dispose d'une machine M, qui sait exécuter en un temps T un programme P s'attaquant à N éléments. On remplace maintenant M par

M', machine 1000 fois plus rapide. Quelle est la taille N' que sait traiter cette nouvelle machine, toujours dans le temps T, dans les cas suivants :

- l'algorithme de P est en $O(\ln n)$;
- l'algorithme de P est en $O(n)$;
- l'algorithme de P est en $O(n \ln n)$;
- l'algorithme de P est en $O(n^2)$;
- l'algorithme de P est en $O(n^3)$;
- l'algorithme de P est en $O(n^5)$;
- l'algorithme de P est en $O(2^n)$;
- l'algorithme de P est en $O(3^n)$.

3.7.2 Évaluation de la complexité d'algorithmes

Le tableau suivant rassemble les mesures de durées d'exécution (en millisecondes) de deux programmes, appliqués à des données de tailles 10, 20, 40 et 80 :

Tailles	10	20	40	80
Algorithme 1	2.0056	6.6161	25.273	92.578
Algorithme 2	4.3994	10.605	24.355	54.843

Déterminez les complexités des algorithmes utilisés. Évaluez les durées d'exécution de ces programmes lorsqu'ils sont appliqués à des données de taille 160.

3.7.3 Un peu de calcul

Calculez l'écart type de la variable aléatoire X, dont un tirage fournit les valeurs suivantes :

```
-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
```

Rappelons que $\sigma^2 = \sum \frac{x_i^2}{n} - (\sum \frac{x_i}{n})^2$. (Valeur attendue : 0.2226357).

3.7.4 Sous-séquence maximale

Étant donné un vecteur V de N éléments, dont on se propose de rechercher la sous-séquence maximale, c'est-à-dire les deux valeurs n et p, avec $n \leq p$, telles que $\sum_{i=n}^{p} V_i$ soit maximale. Si toutes les valeurs de V sont positives ou nulles, le couple $(1, N)$ répond bien évidemment à la question. Ce n'est plus le cas quand V comporte des valeurs négatives. Dans l'exemple suivant :

```
-->V=[31 -49 59 26 -53 58 97 -93 -23 84]
```

la sous-séquence maximale est définie par le couple $(3, 7)$.

Écrire un programme qui recherche la sous-séquence maximale d'un vecteur. Quelle est sa complexité ? Pouvez-vous trouver un algorithme en $O(n)$?

Chapitre 4

Les tableaux

4.1 Scilab, un langage de calcul matriciel

Scilab a été spécifiquement conçu comme environnement de calcul scientifique. Le langage, on l'a vu, permet de définir des matrices au nombre arbitraire de lignes et de colonnes. Ses opérations offrent la particularité de s'appliquer aussi bien à des éléments simples, les *scalaires*, qu'à des vecteurs ou des matrices. Ainsi :

```
-->z=[2 3; 5 7]
 z =
! 2. 3. !
! 5. 7. !
-->sin(z)
 ans =
!   0.9092974   0.1411200 !
! - 0.9589243   0.6569866 !
```

L'expression `[2 3 ; 5 7]` construit une matrice de deux lignes et de deux colonnes. La première ligne contient les valeurs `2` et `3`, la seconde les valeurs `5` et `7`.

La fonction *sinus*, notée `sin` dans le langage, a un *comportement scalaire* : elle s'applique à chaque élément individuel de son paramètre.

Au contraire, la fonction *produit*, représentée par le symbole `*`, a un comportement matriciel : elle calcule le produit de deux matrices :

```
-->sin(z)*sin(z)
 ans =
!   0.6914984  0.2210340 !
! - 1.5019478  0.2963080 !
-->sin(z).*sin(z)
 ans =
!  0.8268218  0.0199149 !
!  0.9195358  0.4316314 !
```

Pour obtenir, comme dans le deuxième exemple, le produit terme à terme des éléments des deux matrices, il faut utiliser le symbole `.*` qui représente cette opération.

De nombreuses opérations matricielles, de la plus simple à la plus complexe, sont directement disponibles dans le langage. C'est la cas de la *transposition*, représentée par le symbole `'` :

```
-->mat=[1,2,3 ;4,5,6]
 mat   =
! 1. 2. 3. !
! 4. 5. 6. !
-->mat'
 ans   =
! 1. 4. !
! 2. 5. !
! 3. 6. !
```

Formellement, cette opération représente la *transposée de la conjuguée* ; l'opération de transposition simple est représentée par la suite `.'`. La *conjuguée* d'une matrice s'obtient par la fonction `conj`.

Les vecteurs étant, dans le langage, représentés par des matrices uni-lignes (ce sont des *vecteurs ligne*), la transposée fournit une matrice uni-colonne, dite aussi *vecteur colonne* :

```
-->w=[1,2,3,4,5]
 w   =
!  1.  2.  3.  4.  5. !
-->w'
 ans =
!  1. !
!  2. !
!  3. !
!  4. !
!  5. !
-->w'*w
 ans =
!   1.   2.   3.   4.   5.  !
!   2.   4.   6.   8.   10. !
!   3.   6.   9.   12.  15. !
!   4.   8.   12.  16.  20. !
!   5.   10.  15.  20.  25. !
-->w*w'
 ans =  55.
```

Les valeurs w et w' étant de dimensions 1×5 et 5×1, les résultats $w' \times w$ et $w \times w'$ sont donc de tailles 5×5 et 1×1 respectivement.

Tout comme dans le cas de la multiplication, plusieurs opérations de *division* sont disponibles :

```
-->v=[1 2 3]
 v =
! 1. 2. 3. !
-->v1 = 1 . / v
 v1 =
! 1. 0.5 0.3333333 !
-->v2 = 1. / v
 v2 =
! 0.0714286 !
! 0.1428571 !
! 0.2142857 !
```

La première de ces expressions calcule l'inverse, terme à terme, de chaque élément du vecteur `[1 2 3]`. La suite des deux caractères `.` et `/` (même séparés par un blanc !), représente l'opération *inverse terme à terme*, alors que le caractère `/`représente la *division matricielle*.

La seconde expression, dans laquelle le point fait partie du nombre, fournit une matrice `v2`, de trois lignes. Le produit matriciel du vecteur `v` par cette matrice fournit la matrice unité :

```
-->v * v2
ans =   1.
```

L'utilisation du *blanc* peut également être une source subtile d'erreurs. Un *vecteur* (qui est en fait une matrice d'une ligne et de n colonnes), peut se noter sous la forme d'une liste d'éléments séparés par des blancs ou des *virgules*, et placée entre *crochets*.

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

Dans le premier exemple, les signes `-` accolés aux nombres font partie intégrante de ceux-ci. Le résultat est un vecteur de 5 éléments. Dans le second exemple, le signe `-` séparé du nombre qu'il précède est reconnu comme l'opérateur de *soustraction*; le vecteur n'a plus que quatre éléments, le premier étant la différence `1-2`. Dans le dernier exemple, le problème est corrigé par l'utilisation de la virgule comme séparateur.

Les *crochets* permettent également la création de matrices. Les éléments sont fournis sous la forme de lignes, séparées par des *points-virgules*. Chaque ligne doit naturellement comporter le même nombre de valeurs.

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

On notera que lors de la définition d'une matrice, le passage à la ligne sur le terminal permet d'indiquer le début d'une nouvelle ligne de la matrice[1]. L'expression ci-dessus est équivalente à :

```
-->[1,2,3
--> 4,5,6]
ans =
! 1. 2. 3. !
! 4. 5. 6. !
```

Présentons encore la fonction `size`, qui fournit les *dimensions* de son paramètre sous la forme d'un couple d'éléments (nombre de lignes et de colonnes) :

```
-->size(3)
ans =
! 1. 1. !
-->size([1,2,3])
ans =
! 1. 3. !
--> size([1,2,3 ;4,5,6])
ans =
! 2. 3. !
```

4.2 Quelques opérateurs élémentaires

Le tableau 4.1, page 73, décrit les opérateurs abordés dans ce chapitre. La colonne «*position*» précise si cet opérateur est *préfixe* (c'est à dire placé devant son opérande unique), *infixe* (placé entre ses opérandes), ou *suffixe* (placé derrière son opérande unique). L'opération \ est décrite plus en détail au § 5.6.2.

4.2.1 Note sur les opérations scalaires

Une opération scalaire est donc une opération qui s'applique élément par élément sur son ou ses paramètres. Dans le cas d'une opération `op` s'appliquant à deux paramètres (comme `+`, `-`, `.*`, etc), le calcul `A op B` est valide dans les cas suivants :

- `A` et `B` ont même structure, c'est-à-dire même nombre de lignes et même nombre de colonnes ;
- l'un des `A` ou `B` est un scalaire (donc, pour SCILAB, un tableau de une ligne et une colonne).

Si ces conditions ne sont pas satisfaites, SCILAB signalera une erreur, et interrompra le calcul courant.

[1] Autrement dit, à l'intérieur des crochets, le passage à la ligne équivant à un point-virgule.

Opération	***Position***	***Description***
`-`	préfixe	*opposée* élément par élément
`+`	infixe	*addition* terme à terme
`*`	infixe	*produit matriciel*
`.*`	infixe	*produit élément par élément*
`/`	infixe	*division matricielle*
`\`	infixe	*division matricielle*
`./`	infixe	*division élément par élément*
`.\`	infixe	*division élément par élément*
`'`	suffixe	*transposée de la conjuguée*
`.'`	suffixe	*transposée*

FIG. 4.1 – Opérateurs sur tableaux

Opération	**Description**
`diag(M)`	extraction de la *diagonale* de la matrice M
`linspace(p,q,n)`	création d'un vecteur de n éléments régulièrement espacés entre p et q
`logspace(p,q,n)`	vecteur de n éléments logarithmiquement espacés entre 10^p et 10^q

FIG. 4.2 – Opérations de création de vecteurs

4.3 Quelques fonctions élémentaires

4.3.1 Création de vecteurs et matrices

Le langage dispose de nombreuses fonctions de création de vecteurs ou de matrices dotées de propriétés spécifiques. À côté de fonctions très spécialisées (comme `inv_coeff`), ou très exotiques (comme `testmatrix`), certaines sont couramment utilisées dans les programmes.

Le tableau4.2, page 73, décrit les opérations de création de vecteurs les plus utiles, tandis que le tableau 4.3, page 73, fait de même pour les matrices.

Si le but est *simplement* de créer un tableau, qui va servir de *container* pour des valeurs qui y seront ultérieurement placées, on peut convenir d'utiliser pour des vecteurs la fonction `vec()`,

Opération	**Description**
`diag(V)`	création d'une *matrice diagonale*, à partir des éléments du vecteur V
`eye(n,p)`	*matrice identité* de taille $n \times p$
`matrix(A,n,p)`	*restructuration* du tableau A selon les dimensions n et p.
`ones(n,p)`	*matrice* de taille $n \times p$, initialisée avec des *1*
`rand(n,p)`	*matrice* de taille $n \times p$, initialisée avec des valeurs aléatoires entre 0 et 1
`tril(A)`	*matrice triangulaire inférieure* construite à partir de A
`triu(A)`	*matrice triangulaire supérieure* construite à partir de A
`zeros(n,p)`	*matrice* de taille $n \times p$, initialisée avec des 0

FIG. 4.3 – Opérations de création de matrices

Opération	Description
`diag(A)`	*diagonale* principale de la matrice A
`pertrans(A)`	*transposée* de A selon la seconde diagonale
`A.'`	*transposée* de A selon la diagonale principale

FIG. 4.4 – Opérations globales sur tableaux

déjà présentée (§ 3.1.2.1), et une fonction similaire pour des matrices, que nous pourrions ainsi définir :

```
function r=mat(x,y)
r=zeros(x,y)
```

Si par contre la matrice *doit* initialement contenir des zéros ou des uns, on préférera à `M=mat(n,p)` des écritures telles que `M=zeros(n,p)` ou `M=ones(n,p)`, qui insistent de manière effective sur la valeur initiale des éléments de la matrice.

On notera que la fonction `eye()` de création de matrice identité permet de créer des matrices rectangulaires : la matrice identité de taille $\min(n, p)$ est complétée avec des zéros pour fournir le résultat :

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

On notera également que la fonction `matrix` de restructuration d'un tableau exige que la taille du tableau à restructurer soit strictement égal au produit $n \times p$. Par ailleurs, cette restructuration s'effectue par remplissage successif des colonnes du résultat :

```
-->matrix(1 :15,3,5)
ans =
! 1.  4.  7.  10. 13.!
! 2.  5.  8.  11. 14.!
! 3.  6.  9.  12. 15.!
```

Enfin, il a été question de «matrice aléatoire» à propos de la fonction `rand()`. Nous utiliserons cette opération chaque fois que nous aurons besoin d'obtenir des nombres aléatoires ; un tel nombre peut être obtenu sous la forme d'une matrice de taille 1×1 par l'expression `rand(1,1)`. Le § **??** aborde en détail la description du générateur de nombres aléatoires en SCILAB, ainsi que les fonctions associées.

4.3.2 Manipulation des matrices

Plusieurs opérations permettent la manipulation globale de matrices. Les plus utiles sont décrites dans la table 4.4, page 74. Quelques exemples d'utilisation :

```
-->M=matrix(1 :9,3,3)
 M =
! 1. 4. 7. !
! 2. 5. 8. !
! 3. 6. 9. !
-->diag(M)
 ans =
! 1. !
! 5. !
! 9. !
-->pertrans(M)
ans =
! 9. 8. 7. !
! 6. 5. 4. !
! 3. 2. 1. !
```

4.3.3 Opérations sur les éléments des matrices

4.3.3.1 Indexation à deux indices

L'opération la plus simple pour l'accès aux éléments d'une matrice est l'*indexation*. Sa syntaxe est : `M(n,p)` ; cette opération lit l'élément de la matrice situé à la ligne n et à la colonne p. Cette désignation est également acceptée dans l'opération d'affectation pour modifier un élément : `M(n,p)=` *expression* :

```
-->M=matrix(1 :15,3,5)
M =
! 1. 4. 7. 10. 13. !
! 2. 5. 8. 11. 14. !
! 3. 6. 9. 12. 15. !
-->M(1,1)
ans = 1.
-->M(2,3)
ans = 8.
-->M(2,3)=111
M =
!  1.   4.   7.   10.  13. !
!  2.   5.   111. 11.  14. !
!  3.   6.   9.   12.  15. !
-->M(2.3, 3.17)
 ans = 111.
```

On notera, comme le montre le dernier exemple, que l'opération d'indexation autorise l'usage d'indices non entiers (ils sont en fait ramenés à l'entier immédiatement inférieur).

Notons aussi que, dans le contexte d'une indexation, la *variable spéciale* `$` représente la valeur maximale permise pour un indice dans cette dimension. Ainsi, avec la variable `M` ci-dessus, `M(1,$)` représente le cinquième élément de la première ligne, `M($,3)` le troisième élément de dernière ligne, `M($-1, $/2)` l'élément d'indice `(2,2.5)`, etc :

```
-->M(1,$)
 ans = 13.
-->M($,3)
 ans = 9.
-->M($-1, $/2)
 ans = 5.
```

4.3.3.2 Indexation à un indice

L'indexation propose une autre forme, dans laquelle un seul indice est utilisé. Dans ce cas, l'objet indicé est considéré comme un objet linéaire, arrangé par colonnes : $M(1) \equiv M(1,1)$, $M(2) \equiv M(2,1)$, $M(3) \equiv M(3,1)$, etc., et $M(p) \equiv M(p,1)$ pour un vecteur colonne. Pour un tableau quelconque, de dimensions $P \times Q$, $M(p,q) \equiv M(p + (q-1) * P)$. Sous cette forme, la variable `$` a comme valeur le nombre total d'éléments de M. On notera que c'est cette forme d'indexation qui a été utilisée dans tout le chapitre 3, où nous manipulions uniquement des vecteurs.

4.3.3.3 Indexations par des vecteurs

Les modes d'indexation permis en Scilab sont en réalité bien plus nombreux et complexes que les paragraphes précédents ne le laissent entendre.

Une première idée de l'indexation matricielle est que les indices (lignes et colonnes) peuvent être des vecteurs, ce qui autorise l'extraction de sous-ensembles rectangulaires d'une matrice. Considérons la matrice suivante :

```
-->M=matrix(1 :15,3,5)
M =
! 1. 4. 7. 10. 13. !
! 2. 5. 8. 11. 14. !
! 3. 6. 9. 12. 15. !
```

Nous allons pouvoir extraire des sections de ce tableau, par exemple les colonnes 1, 4 et 5 des lignes 2 et 3 :

```
-->M([2 3],[1 4 5])
 ans =
! 2. 11. 14. !
! 3. 12. 15. !
```

Dans le contexte d'une indexation, la variable $ peut naturellement être utilisé pour construire les vecteurs qui vont servir à indexer une matrice :

```
-->M(2 :$,3 :$)
 ans =
! 8. 11. 14. !
! 9. 12. 15. !
```

On notera enfin que l'écriture « :» peut servir à remplacer «1 :$», et signifie «tous les éléments le long de cette dimension» :

```
-->M(2 :$, :)
 ans =
! 2. 5. 8. 11. 14. !
! 3. 6. 9. 12. 15. !
```

Enfin, l'indexation permet de même de modifier les sous-parties d'un tableau :

```
-->V=1 :10
 V =! 1. 2. 3. 4. 5. 6. 7. 8. 9. 10. !
-->V([4 6 7])= [100 101 102]
 V =! 1. 2. 3. 100. 5. 101. 102. 8. 9. 10. !
-->V([4 6 7])= 0
 V =! 1. 2. 3. 0. 5. 0. 0. 8. 9. 10. !
```

Dans ce dernier exemple, l'opérande droit de l'affectation est étendu aux dimensions nécessaires. Il est possible d'affecter de cette manière une valeur à tous les éléments d'un vecteur :

```
-->V( :)=0
 V =! 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. !
```

Note : dans l'écriture ci-dessus, le vecteur `V` concerné est naturellement celui de l'exemple précédent. Cependant, l'affectation à un tableau inexistant entraîne la création automatique de ce tableau, pourvu d'un nombre suffisant d'éléments pour recevoir les valeurs affectées :

```
-->X([3 5 8])=[1 2 3]
 X =! 0. 0. 1. 0. 2. 0. 0. 3. !
-->Y( :)=0
 Y = []
```

4.3.3.4 Création et suppression d'éléments

L'indexation, enfin, va nous permettre de modifier les dimensions d'un vecteur, en y insérant ou supprimant des éléments. L'expression *V(k)=v* a pour effet, si le vecteur est de taille inférieure à k, d'étendre ce vecteur jusqu'à la taille k, puis d'affecter la valeur v au *k-ème* élément de V.

```
-->V=1 :5
 V =! 1. 2. 3. 4. 5. !
-->V(8)=88
 V =! 1. 2. 3. 4. 5. 0. 0. 88. !
-->M=matrix(1 :15,3,5)
 M =
! 1. 4. 7. 10. 13. !
! 2. 5. 8. 11. 14. !
! 3. 6. 9. 12. 15. !
-->M(4,7)=666
 M =
! 1.  4.  7.  10. 13. 0.  0.   !
! 2.  5.  8.  11. 14. 0.  0.   !
! 3.  6.  9.  12. 15. 0.  0.   !
! 0.  0.  0.  0.  0.  0.  666. !
```

Par ailleurs, l'affectation du tableau vide à un ensemble d'éléments d'un tableau a pour effet de supprimer les éléments de ce tableau.

```
-->K
 K =
!  1.  2.  3.  4.  5. !
!  6.  7.  8.  9. 10. !
-->K( :,4)=[]
 K =
!  1.  2.  3.  5. !
!  6.  7.  8. 10. !
```

4.4 Quelques exercices

4.4.1 Un exemple : l'algorithme du carré magique

Prenons un exemple qui va nous permettre de manipuler des éléments de matrices : la création d'un carré magique.

Description Un *carré magique* est une matrice numérique carrée, dont la somme des éléments selon les lignes, les colonnes ou les diagonales est toujours égale à une même valeur. Le tableau ci-dessous constitue un carré magique d'ordre trois, construit selon l'algorithme qui va être exposé.

8	1	6
3	5	7
4	9	2

Nous utiliserons l'algorithme suivant :

- cet algorithme permet la création de carré magique d'ordre n impair.
- il consiste à déposer dans la matrice les nombres consécutifs de 1 à n^2.
- la première case à remplir est celle du milieu de la première ligne.
- quand une case a été remplie, la case suivante est celle qui est située à la ligne précédente et à la colonne suivante (modulo la taille de la matrice) ; si cette case est déjà occupée, on choisit la case qui est dans la même colonne que celle qui vient d'être remplie, et qui est située à la ligne suivante.

On peut vérifier que le carré magique ci-dessus a bien été composé en suivant cet algorithme.

Programmation de l'algorithme du carré magique Elle ne devrait pas poser de problème. En cas de difficulté, on pourra consulter une solution possible du problème page **??**.

Test du carré magique On se propose d'écrire maintenant une fonction qui vérifie que son paramètre est un carré magique bien formé. Il faut donc vérifier que les sommes le long des lignes, colonnes et diagonales sont bien égales. Peut-on éviter les boucles, en appliquant des opérations matricielles ?

4.5 L'application vectorielle

L'application vectorielle est une opération qui permet d'appliquer une fonction f à chaque élément d'un tableau T. Le résultat est un tableau R, tel que $R_i = f(T_i)$.

Il se trouve qu'en SCILAB, la plupart des opérations scalaires (les opérateurs, comme `*`, `<`, `<=` et les autres, les fonctions du système, etc.) étudiées au chapitre 2 pratiquent implicitement cette application vectorielle :

```
-->sin(0.1)
 ans = 0.0998334
-->sin([0.1 0.2 0.3 ; 0.4 0.5 0.6])
 ans =
! 0.0998334 0.1986693 0.2955202 !
! 0.3894183 0.4794255 0.5646425 !
-->[1 2 3]+[4 5 6]
 ans =! 5. 7. 9. !
```

Si les fonctions que nous programmons utilisent ces opérations scalaires, elles s'appliqueront donc implicitement à tout tableau :

```
-->deff('r=fun(x)', 'r=sin(x)+cos(x)')
-->fun(0.1)
 ans = 1.0948376
-->fun([0.1 0.2 0.3 ; 0.4 0.5 0.6])
 ans =
! 1.0948376 1.1787359 1.2508567 !
! 1.3104793 1.3570081 1.3899781 !
```

Dans certains cas cependant, l'algorithme de la fonction, parce qu'il utilise des opérations nécessitant des éléments simples, ne peut pas s'appliquer «automatiquement» à des tableaux quelconques :

```
-->deff('r=fact(n)', 'r=prod(1 :n)')
-->fact(5)
ans = 120.
-->fact([2 3 5])
  !--error 204
Argument 2 of ' :' : wrong type argument, expecting a scalar
at line 2 of function fact
called by : fact([2 3 5])
```

Il peut être utile de passer par une procédure qui va appliquer la fonction à chaque élément d'un tableau. Des procédures de ce genre existent dans dans langages tels Lisp, APL, etc. Nous allons écrire la notre :

```
function R=map(f, A)
R=zeros(A)
for k=1 :count(A)
    R(k)=f(A(k))
end
```

Voici son utilisation dans le cas litigieux ci-dessus :

```
-->map(fact, [2 3 5])
ans =! 2. 6. 120. !
```

Il est clair que, de manière générale, il vaut mieux utiliser les primitives existantes, et tenter si possible d'obtenir une procédure qui s'applique implicitement à des tableaux de taille quelconque (si le besoin risque de s'en faire sentir) ; dans le cas contraire, il peut être intéressant de définir une fonction s'appliquant correctement à un élément simple, puis d'utiliser une opération telle que `map()` pour étendre l'application de la fonction à des tableaux quelconques. Ainsi, à cette définition d'une fonction qui rend le signe (*-1*, *0* ou *1*) de son argument :

```
function z=signe(x)
if x>0 then z=1 ; elseif x==0 then z=0 ; else z=-1 ; end
```

on préférera une écriture telle que la suivante, qui fournit une opération s'appliquant sur des tableaux quelconques :

```
function z=signe(x)
z=x./(abs(x)+(x==0))
```

(dans cette dernière formulation, le résultat de `(x==0)`, de type *booléen*, est converti dans le type *numérique* afin de pouvoir réaliser l'addition, les *T* devenant des *1* et les *F* des *0*)

4.6 Notes sur la mise au point des programmes

Voici la suite des notes sur la mise au point des programmes.

4.6.1 Variables d'une fonction

Une fonction définie (dite *macro*, dans la terminologie SCILAB) est un programme, avec des *entrées* (les *paramètres*), des *sorties* (les *résultats*), et un *code source*, consistant en une séquence d'instructions écrites en SCILAB. Dans une fonction, toutes les variables utilisées en écriture (c'est à dire, les variables qui, à un moment donné ou à un autre, reçoivent une valeur), sont *locales* à la fonction. Lors de l'exécution de la fonction, *ces variables prennent comme valeur une copie de la valeur de la variable externe définie dans l'espace de travail.*

La fonction `macrovar()` accepte comme paramètre une fonction définie, et fournit une liste de 5 éléments qui représentent respectivement :

- les variables d'entrée de la fonction ;
- les variables de sortie de la fonction ;
- les variables globales utilisées par la fonction ;
- les fonctions appelées par la fonction ;
- les variables locales de la fonction.

Cet ensemble d'informations peut s'avérer utile pour détecter certaines anomalies (par exemple, une faute de frappe crée une variable locale nouvelle, qui ne sera jamais utilisée, alors que la «vraie» variable locale ne va pas recevoir la valeur que le programmeur voulait lui affecter). Prenons comme exemple la fonction suivante, dont le rôle est de «tabuler» une fonction à deux paramètres :

```
function Z=mat1(n,p,f)
Z=zeros(n,p) ;
for x=1 :n do
    for y=1 :p do
        Z(x,y)=f(x,y) ;
    end
end
endfunction
```

L'opération `macrovar()` nous fournit les cinq résultats décrits ci-dessus :

```
-->macrovar(mat1)
ans =
ans(1) !n p f !
ans(2) Z
ans(3) []
ans(4) zeros
ans(5) !x y !
```

4.7 Exercices

4.7.1 Lecture

Allez consulter la documentation de chaque opération présentée dans ce chapitre.

4.7.2 Notation

Note : à résoudre sans machine ! Quel est le résultat de :

```
-->[1  1+i  1+2i  1+3i]
```

Quel est le résultat de :

```
-->[1  2  3 ...
--> 4  5  6]
```

4.7.3 Construction de matrice

Trouver une expression simple qui construit la matrice suivante :

```
! 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. !
```

4.7.4 Oeil gauche

La fonction `eye()` utilisée avec deux paramètres fournit une matrice identité de la taille précisée :

```
-->eye(2,6)
 ans =
! 1. 0. 0. 0. 0. 0. !
! 0. 1. 0. 0. 0. 0. !
-->eye(4,4)
ans =
! 1. 0. 0. 0. !
! 0. 1. 0. 0. !
! 0. 0. 1. 0. !
! 0. 0. 0. 1. !
```

Nous aimerions disposer de la fonction symétrique, que nous baptiserons lefteye(), qui nous fournisse le miroir (horizontal) des résultats obtenus avec eye :

```
-->lefteye(2,6)
 ans =
! 0. 0. 0. 0. 0. 1. !
! 0. 0. 0. 0. 1. 0. !
-->lefteye(4,4)
```

```
ans =
! 0. 0. 0. 1. !
! 0. 0. 1. 0. !
! 0. 1. 0. 0. !
! 1. 0. 0. 0. !
```

Comment peut-on, en utilisant cette fonction, obtenir le miroir d'une matrice carrée ?

4.7.5 Matrices de Hilbert

Écrire la fonction `hilbert(n)` qui fournit une matrice de *Hilbert*, c'est à dire une matrice carrée telle que $M_{i,j} = \frac{1}{i+j-1}$.

4.7.6 Petit utilitaire

Écrire la fonction `ravel(X)`, qui, appliquée à un tableau `X` de taille quelconque, fournit le vecteur ayant le même nombre d'éléments que le tableau. Les éléments sont pris dans l'ordre des colonnes du tableau `X` :

```
-->ravel([1 2 3 4 5])
 ans =! 1. 2. 3. 4. 5. !
-->ravel(3)
 ans = 3.
-->ravel([1 2 3 ;4 5 6])
 ans =! 1. 4. 2. 5. 3. 6. !
-->ravel([])
 ans = []
```

4.7.7 Produit scalaire

– Écrire une fonction calculant le produit scalaire de deux vecteurs $\overrightarrow{A}$ et $\overrightarrow{B}$:

$$\overrightarrow{A} \bullet \overrightarrow{B} = \sum_{i=1}^{n} a_i b_i$$

– Écrire une fonction qui calcule l'angle α de deux vecteurs $\overrightarrow{A}$ et $\overrightarrow{B}$ au moyen de la formule :

$$\cos\alpha = \frac{\overrightarrow{A} \bullet \overrightarrow{B}}{\sqrt{(\overrightarrow{A} \bullet \overrightarrow{A}) \times (\overrightarrow{B} \bullet \overrightarrow{B})}}$$

4.7.8 Tri ascendant

La fonction `sort()` nous permet de trier un vecteur par valeurs descendantes de ses éléments :

```
-->v=[9,2,7,4,5,6,0,3,8,1]
v =! 9. 2. 7. 4. 5. 6. 0. 3. 8. 1. !
-->sort(v)
ans =! 9. 8. 7. 6. 5. 4. 3. 2. 1. 0. !
```

Écrire la fonction `rsort()` qui opère de manière inverse, c'est-à-dire qu'elle fournit le vecteur trié par valeurs ascendantes :

```
-->rsort(v)
ans =! 0. 1. 2. 3. 4. 5. 6. 7. 8. 9. !
```

4.7.9 Réversion d'un vecteur

Écrire une nouvelle version, plus performante, de l'opération de réversion (c.f. §3.2.2, page 53), utilisant les opérations présentées dans ce chapitre. On utilisera la fonction `chrono` (A.2.2) pour évaluer les performances des différents programmes.

Chapitre 5

Les tableaux, suite

Ce second chapitre consacré aux tableaux va nous permettre d'en approfondir la manipulation.

5.1 Le balayage

Le *balayage* (*scan*, en anglais) est une opération qui permet de décrire les éléments d'un vecteur au moyen d'une fonction s'appliquant à deux éléments, à la manière de la réduction. Le balayage fournit cependant un résultat vectoriel, qui contient les *résultats des réductions*[1] *successives des éléments* 1 *à* k *du vecteur*. Formellement,

$$R = scan(f, V) \Rightarrow R_k = reduc(f, V(1:k))$$

Le langage SCILAB fournit deux opérations appliquant cette définition, `cumsum` et `cumprod`, qui réalisent respectivement des *sommes* et *produits cumulatifs* :

```
-->V=[1 2 3 4 5 6]
 V =! 1. 2. 3. 4. 5. 6. !
-->cumprod(V)
 ans =! 1. 2. 6. 24. 120. 720. !
-->cumsum(V)
 ans =! 1. 3. 6. 10. 15. 21. !
```

Il est naturellement possible, à l'instar de ce qui a été fait pour la réduction, de proposer une opération `scan`, qui prend comme paramètres un vecteur, une fonction et un élément neutre pour cette fonction, et réalise le balayage du vecteur par cette fonction. Voici une version possible de cette opération :

```
function Z=scan(f,V)
S=V(1)
Z=V
N=count(V)
```

[1] c.f. § 3.5.1, page 63.

```
for k=2 :N do
    S=f(S,V(k)) ;
    Z(k)=S ;
end
```

Quelques exemples, qui nous permettent de retrouver l'équivalent des opérations `cumsum` et `cumprod`, au moyen de deux fonctions définies, `add()` et `mul()` déjà présentées :

```
-->scan(add, [1 2 3 4 5])
 ans =! 1. 3. 6. 10. 15. !
-->scan(mul, [1 2 3 4 5])
 ans =! 1. 2. 6. 24. 120. !
```

Des approximations successives de e par la somme des inverses des factorielles des premiers entiers :

```
-->scan(add, 1 ./ [1, scan(mul, 1 :10)])
ans =! 1. 2. 2.5 2.6666667 2.7083333 2.7166667 2.7180556
           2.718254 2.7182788 2.7182815 2.7182818 !
```

5.2 Le produit externe

Le *produit externe* est une opération qui s'applique à deux tableaux et à une fonction à deux arguments, et qui fournit le résultat de l'application de la fonction sur chaque paire constituée d'un élément du premier tableau et d'un élément du second tableau. L'opération en elle-même est aisée à programmer :

```
function Z=outer(f,A,B)
NA=count(A)
NB=count(B)
Z=zeros(NA,NB)
for a=1 :NA do
    for b=1 :NB do
        Z(a,b)=f(A(a),B(b)) ;
    end
end
```

Quelques exemples :

```
-->outer(add, 1 :3, 1 :5)
 ans =
! 2. 3. 4. 5. 6. !
! 3. 4. 5. 6. 7. !
! 4. 5. 6. 7. 8. !
-->deff('z=fun(a,b)', 'z=1/(a+b-1)')
```

```
-->outer(fun, 1 :3, 1 :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!
```

Cette dernière expression nous fournit la matrice de Hilbert. Voici encore l'équivalent de «`eye`» :

```
-->deff('z=eql(a,b)', 'z=0+(a==b)')
-->outer(eql, 1 :3, 1 :5)
ans =
! 1. 0. 0. 0. 0.!
! 0. 1. 0. 0. 0.!
! 0. 0. 1. 0. 0.!
```

Deux fonctions d'apparence anodines, `gauche` et `droit`, qui fournissent respectivement leurs opérandes *gauche* et *droit*, nous permettent de construire agréablement certaines matrices :

```
-->deff('z=gauche(a,b)', 'z=a')
-->deff('z=droit(a,b)', 'z=b')
-->outer(gauche, 1 :3, 1 :5)
 ans =
! 1. 1. 1. 1. 1.!
! 2. 2. 2. 2. 2.!
! 3. 3. 3. 3. 3.!
-->outer(droit, 1 :3, 1 :5)
ans =
! 1. 2. 3. 4. 5.!
! 1. 2. 3. 4. 5.!
! 1. 2. 3. 4. 5.!
```

On notera que la formulation que nous avons adoptée pour l'opération `outer` impose un type numérique comme résultat. En fait, en supprimant simplement la ligne qui prédéfinit le résultat (nous nommerons `outer1` cette nouvelle écriture), on peut s'affranchir de cette contrainte, et obtenir par exemple des résultats de type booléen :

```
-->deff('z=eql(a,b)', 'z=a==b')
-->outer1(eql, 1 :3, 1 :5)
 ans =
! T F F F F!
! F T F F F!
! F F T F F!
```

5.2.1 Exercice

La suppression de la ligne «`Z=zeros(NA,NB)`» n'est pas aussi anodine que l'on peut le penser. Voici les temps d'exécution des deux versions sur des opérandes de grande taille :

```
-->chrono('outer(add, 1 :500, 1 :500)')
14110 ms.
-->chrono('outer1(add, 1 :500, 1 :500)')
32050 ms.
```

À votre avis, à quoi sont dues ces différences de temps d'exécution ? Comment écrire notre opération, pour qu'elle aille aussi vite que `outer`, mais ne soit pas contrainte à ne fournir que des résultats numériques ?

5.3 Compression

La *compression* est l'opération qui permet de sélectionner les éléments d'un tableau à partir d'un vecteur booléen, les valeurs *vrai* correspondant aux éléments à conserver, les valeurs *faux* aux éléments à rejeter. La compression de V par le vecteur booléen B se note *V(B)* :

```
-->V=[2 7 3 -1 5 -3 8 4 -1 -2]
 V =! 2. 7. 3. - 1. 5. - 3. 8. 4. - 1. - 2. !
-->B=[T T T T T F F T F T]
 B =! T T T T T F F T F T!
-->V(B)
 ans =! 2. 7. 3. - 1. 5. 4. - 2. !
```

Cette opération de compression va nous permettre de sélectionner les éléments d'un vecteur qui répondent à un certain critère, par exemple les éléments positifs, ceux qui sont impairs, etc :

```
-->V(V>0)
 ans =! 2. 7. 3. 5. 8. 4. !
-->V(0<>modulo(V,2))
 ans =! 7. 3. - 1. 5. - 3. - 1. !
-->V(abs(V)<4)
 ans =! 2. 3. - 1. - 3. - 1. - 2. !
```

5.4 Le produit interne

Le *produit interne* est une opération qui s'applique à deux tableaux et deux fonctions à deux arguments, et qui fournit le résultat de la réduction, par la première fonction, de l'application de la seconde fonction sur chaque paire constituée d'un élément du premier tableau et d'un élément du second tableau. L'opération en elle-même est aisée à programmer pour des vecteurs :

```
function Z=inner(f,g,A,B)
NA=count(A)
NB=count(B)
check(NA==NB & NA>0, "Opérandes incompatibles")
Z=g(A(1),B(1))
for k=2 :NA do
    Z=f(Z, g(A(k),B(k)))
end
```

L'opération, utilisée avec l'addition et la multiplication telles que définies ci-dessus, nous fournit le produit scalaire :

```
-->inner(add,mul,[2 4 5],[3 5 1])
 ans = 31.
-->[2 4 5]*[3 5 1]'
 ans = 31.
```

Naturellement, d'autres opérations utiles peuvent être réalisées au moyen de la fonction `inner`.

5.5 Itération et vectorisation

Nous avons présenté au chapitre précédent l'utilitaire `map()` (§ 4.5), qui permet l'application d'une fonction aux différents éléments d'un vecteur. Telle que programmée, cette fonction s'applique à un tableau de taille quelconque, et fournit un résultat de même taille :

```
-->tab=[1 2 2.3 4 2.3 ; 0 0.1 2.1 2.3 3.2]
 tab =
! 1.  2.   2.3  4.   2.3 !
! 0.  0.1  2.1  2.3  3.2 !
-->res=map(sinus,tab)
 res =
!  0.8414710    0.9092974    0.7457052 - 0.7568025   0.7457052   !
!  0.           0.0998334    0.8632094   0.7457052 - 0.0583741   !
```

(Dans cet exemple, `sinus` est une fonction qui calcule simplement le *sinus* de son argument.) Naturellement, il est *toujours* préférable d'utiliser directement l'opération prédéfinie lorsque celle-ci s'applique à des vecteurs :

```
-->sin(tab)
 ans =
!  0.8414710    0.9092974    0.7457052 - 0.7568025   0.7457052   !
!  0.           0.0998334    0.8632094   0.7457052 - 0.0583741   !
```

En revanche, *on évitera autant que possible toute solution intermédiaire*, dans laquelle par exemple, une boucle serait utilisée pour appliquer la fonction aux aux éléments individuels, comme dans :

```
-->res=tab ;
-->for k=1 :10 do res(k)=sin(tab(k)) ; end
-->res
!  0.8414710    0.9092974    0.7457052 - 0.7568025   0.7457052   !
!  0.           0.0998334    0.8632094   0.7457052 - 0.0583741   !
```

Pourquoi ? Même si cette dernière écriture peut sembler plus performante (0.56 contre 0.70 millisecondes) que l'application `map(sinus,tab)`, elle est aussi beaucoup plus *fragile* : par exemple, le nombre d'éléments du tableau est fixé à 10, ce qui veut dire que si le programme est modifié pour s'appliquer à des tableaux de 12 éléments, seuls les dix premiers auront leur sinus calculé ! Une erreur qui peut être très difficile à repérer dans un gros code de calcul. L'autre écriture, naturellement, s'adapte automatiquement à un changement de taille des opérandes.

5.6 Algèbre linéaire

5.6.1 Outils de base

SCILAB, en tant que logiciel scientifique, dispose naturellement d'un grand nombre d'opérations dans le domaine de l'algèbre linéaire. Nous avons déjà rencontré des opérations comme le produit de matrices, représenté par le symbole *, l'élévation à la puissance, ^, et quelques autres :

```
-->A=rand(2,2)
 A =
!  0.0683740   0.6623569  !
!  0.5608486   0.7263507  !
-->A*A*A*A
 ans =
!  0.3761174   0.6712671  !
!  0.5683933   1.0429453  !
-->A^4
 ans =
!  0.3761174   0.6712671  !
!  0.5683933   1.0429453  !
```

Un problème classique est celui du calcul du *déterminant* d'une matrice. La fonction `det()` fournit le déterminant de son paramètre :

```
-->A
 A =
!  0.0683740   0.6623569  !
!  0.5608486   0.7263507  !
-->det(A)
 ans = - 0.3218184
```

```
-->0.0683740*0.7263507 - 0.5608486* 0.6623569
 ans = - 0.3218184
```

Autre opération classique, *l'inversion de matrice*, obtenue au moyen de la fonction `inv()` :

```
-->inv(A)
 ans =
! - 2.2570201    2.0581696 !
!   1.7427485  - 0.2124615 !
-->A*inv(A)
 ans =
! 1. 0. !
! 0. 1. !
```

Les opérations `rank()` et `norm()` fournissent respectivement le *rang* et la *norme* d'une matrice :

```
-->rank(A)
 ans = 2.
-->norm(A)
 ans = 1.0950625
```

La fonction `trace()` fournit la *trace* d'une matrice :

```
-->trace(A)
ans = 0.7947247
```

Valeurs et *vecteurs propres* peuvent être obtenus par les fonctions `spec()` et `bdiag()` :

```
-->spec(A)
 ans =
! - 0.2952525 !
!   1.0899772 !
-->bdiag(A)
 ans =
! - 0.2952525   0.         !
!   0.          1.0899772 !
```

On peut ainsi vérifier que le déterminant est bien égal au produit des valeurs propres :

```
-->det(A)
 ans = - 0.3218184
-->prod(spec(A))
 ans = - 0.3218184
```

Enfin, il est possible d'effectuer la *triangulation* d'une matrice au moyen de la fonction `htrianr()` :

```
-->htrianr(A)
 ans =
! - 0.3218184   0.9118969 !
!   0           1         !
```

5.6.2 Résolution de systèmes linéaires

SCILAB propose une méthode standard de résolution de système linéaire lorsque la matrice est carrée. La résolution de $Ax = y$ se note $x = A\backslash y$. Dans cette écriture, A est une matrice $n \times n$, et y est une matrice colonne $n \times 1$:

```
-->A=rand(5,5)
 A =
!  0.1985144  0.8833888  0.312642   0.3321719  0.6325745 !
!  0.5442573  0.6525135  0.3616361  0.5935095  0.4051954 !
!  0.2320748  0.3076091  0.2922267  0.5015342  0.9184708 !
!  0.2312237  0.9329616  0.5664249  0.4368588  0.0437334 !
!  0.2164633  0.2146008  0.4826472  0.2693125  0.4818509 !
-->y=rand(5,1)
 y =
! 0.2639556 !
! 0.4148104 !
! 0.2806498 !
! 0.1280058 !
! 0.7783129 !
-->x=A\y
 x =
!   1.9687768 !
! - 0.2866983 !
!   1.4349965 !
! - 2.0543488 !
!   0.5693368 !
-->A*x
ans =
! 0.2639556 !
! 0.4148104 !
! 0.2806498 !
! 0.1280058 !
! 0.7783129 !
```

On notera que l'opération permet de résoudre simultanément plusieurs systèmes linéaires définis par la même matrice, $Ax_1 = y_1$, $Ax_2 = y_2 \ldots Ax_n = y_n$ en écrivant : A\[y_1, $y_2 \ldots$ y_n].

SCILAB propose une autre fonction pour résoudre ce problème, `linsolve()`, qui résout l'équation $Ax + y = 0$. On retrouve la même solution que ci-dessus, en appliquant la fonction à A et à $-y$:

```
-->linsolve(A,-y)
 ans =
```

```
!   1.9687768 !
! - 0.2866983 !
!   1.4349965 !
! - 2.0543488 !
!   0.5693368 !
```

5.7 Plus avant dans Scilab

5.7.1 Variables

L'utilisateur aura compris qu'il existe en SCILAB plusieurs types de variables :
- les variables qui sont définies dans l'*espace de travail*, parce qu'elles ont été créées par une affectation en mode interactif, ou encore dans un script exécuté par SCILAB (tel le fichier `.scilab`, exécuté automatiquement lors du lancement du logiciel) ;
- les *variables locales* à une fonction ; ces variables apparaissent dans l'en-tête de la fonction, en tant que paramètres ou résultats, ou encore font l'objet d'une affectation dans le corps de la fonction ;
- les *variables globales*, enfin, qui font l'objet d'une présentation spécifique au § 5.7.2.

Des règles différentes concernent l'accessibilité, en lecture ou en écriture, de ces variables.

5.7.1.1 Consultation de variables

À tout instant, une fonction peut consulter la valeur d'une variable locale (si une valeur a été affectée à cette variable). Une fonction F peut également consulter les valeurs des variables de l'espace de travail, ou encore des variables locales de fonctions qui font appel à F. La petite session ci-dessous démontre ce mécanisme :

```
-->toto=3
 toto = 3.
-->deff('z=fun()','z=toto')
-->fun()
 ans = 3.
```

Nous définissons ici une fonction `fun()` qui consulte la valeur de la variable `toto`, et fournit cette valeur comme résultat. La valeur de `toto` qui est visible dans le contexte de la fonction `fun()` est la valeur de la variable `toto` de l'espace de travail. Si la fonction `fun()` est appelée par une autre fonction, qui définit elle-même une variable locale de nom `toto`, c'est cette variable locale qui sera visible :

```
-->deff('r=foo()', ['toto=7', 'r=fun()'])
-->foo()
ans = 7.
```

Dans cet exemple, nous dirons que la variable `toto` de l'espace de travail est *masquée*, durant l'exécution de `foo()`, par la variable locale `toto` de cette fonction.

Attention ! L'existence d'une variable locale ne débute que lorsque celle-ci reçoit une valeur. Dans l'exemple suivant, `toto`, bien que variable locale de la fonction `bar()`, ne reçoit de valeur qu'après l'exécution de `fun()`. La valeur de `toto` visible à ce moment là est toujours la valeur de l'espace de travail.

```
-->deff('r=bar()', ['r=fun()', 'toto=7'])
-->bar()
 ans = 3.
```

Plus subtil, dans l'exemple suivant, `toto` est un paramètre de la fonction `baz()`, donc une variable locale de celle-ci :

```
-->deff('r=baz(toto)', 'r=fun()')
-->baz(8)
 ans = 8.
-->baz()
 ans = 3.
```

Lors du premier appel de la fonction `baz()`, `toto` reçoit la valeur du paramètre, donc `8`, et c'est bien cette valeur qui est visible dans la fonction `fun()`. Lors du second appel, aucun paramètre n'est passé à la fonction. La variable locale `toto` est donc indéfinie, et c'est la valeur de l'espace de travail qui est visible lors de l'appel de la fonction `fun()`, d'où le résultat `3`.

5.7.1.2 Modification de variables

Il n'est en principe pas possible de modifier les variables externes à une fonction, dans la mesure où une affectation à une variable, à l'intérieur d'une fonction, rend implicitement locale cette variable.

Cependant, l'opération `return()`, déjà présentée au § 3.6.3, permet de modifier une ou plusieurs variables de l'environnement appelant, comme le montre cet exemple :

```
function R=foo(V)
R=V
disp("R vaut "+string(R))
bar()
disp("R vaut "+string(R))
endfunction
function bar()
R=return(3)
endfunction
```

Lors de l'exécution, les impressions de la valeur de `R` montrent que cette variable a été effectivement modifiée par l'appel de `bar()` :

```
-->foo(5)
R vaut 5
R vaut 3
 ans = 3.
```

Cependant, s'il est nécessaire que des fonctions puissent communiquer entre elles en consultant et modifiant un ensemble de variables, il est préférable de déclarer *globales* ces variables.

5.7.2 Variables globales

Les *variables globales* constituent une classe de variables qui peuvent être accessibles en lecture et en écriture par différentes fonctions. L'espace des variables globales est *différent* de l'espace de travail. Une déclaration spécifique, `global`, permet d'indiquer qu'une variable est globale. L'exemple suivant démontre les caractéristiques de ces objets :

```
-->toto='Voici toto'
 toto = Voici toto
-->deff('vput(V)',['global toto' ; 'toto=V'])
-->deff('Z=vget()',['global toto' ; 'Z=toto'])
-->vput(33)
-->vget()
 ans = 33.
-->vput(45)
-->vget()
 ans = 45.
-->toto
toto = Voici toto
```

La première instruction définit une variable de nom `toto`, qui nous servira de témoin. Les fonctions `vget()` et `vput()` déclarent toutes deux une variable globale de nom `toto` ; l'une a pour action d'affecter la valeur de son paramètre à cette variable, l'autre de fournir comme résultat la valeur de cette variable. Les manipulations suivantes montrent que les fonctions font ce que l'on attend d'elles, la dernière expression montre que la valeur de la variable `toto` de l'espace de travail n'a pas été modifiée.

Les variables globales peuvent être accessibles en mode interactif ; il suffit de les déclarer :

```
-->toto
 toto = Voici toto
-->global toto
-->toto
 toto = 45.
```

On notera que cette déclaration fait disparaître la valeur `toto` de l'espace actif.

Enfin, une variable globale peut servir de mémoire à une fonction. Dans l'exemple ci-dessous, la fonction `counter()` a pour but de fournir une nouvelle valeur à chaque appel (les entiers naturels successifs). Ce résultat est obtenu par l'utilisation d'une variable globale, `counter_value`, qui est incrémentée à chaque appel. Lors de la première déclaration d'une variable globale, celle-ci est initialisée avec la valeur `[]`. La fonction teste donc ce cas particulier, et affecte alors à la variable la valeur 0, afin d'initialiser le processus :

```
function Z=counter()
global counter_value
if isempty(counter_value) then
    counter_value = 0 ;
end
counter_value = counter_value + 1 ;
Z= counter_value ;
endfunction
```

Voici quelques exemples d'utilisation de cette fonction :

```
-->counter()
ans = 1.
-->counter()
ans = 2.
-->2*counter()+10*counter()
ans = 46.
```

5.7.3 Effacement de variables

Signalons encore la fonction `clear` qui permet d'effacer les variables dont les noms sont données en paramètres :

```
-->toto=3 ; titi=5 ;
-->clear toto titi
-->titi
  !--error 4
undefined variable : titi
```

Attention, sans paramètres, la commande efface ***toutes*** *les variables définies au cours de la session.* La commande `clearglobal`, dont la syntaxe et le fonctionnement sont similaires, permet d'effacer les *variables globales.*

5.7.4 Paramètres et résultats de fonctions

Une fonction, on l'a dit, peut fournir plusieurs résultats. Ceci se déclare en utilisant dans l'en-tête une liste de noms, séparés par des virgules, et placée entre crochets, au lieu d'un nom unique. Ceci répond à deux besoins :

- il peut être nécessaire, dans certains cas, de fournir plusieurs résultats pour une procédure ; ces résultats peuvent être de natures ou de dimensions telles qu'il ne soit pas simple de les rassembler en une structure unique (par exemple, un nombre et une chaîne de caractères).
- une fonction peut fournir un résultat «principal», toujours utilisé, et un ou plusieurs résultats secondaires, dont la valeur n'est pas systématiquement utilisée par le programme appelant.

La contrainte est que, lors de l'appel, il faut prévoir une *liste de variables* pour recevoir la *liste des résultats* produite par la fonction :

```
function [u,v]=sincos(a)
u=sin(a)
v=cos(a)
```

Voici deux exemples d'appel :

```
-->[u,v]=sincos(0.135)
 v = 0.9909013
 u = 0.1345903
-->k=sincos(0.135)
 k = 0.1345903
```

La règle est la suivante : si une fonction rend n résultats, et que, lors de l'appel, ces résultats ne sont affectés qu'à p variables (avec $p < n$), seuls les p premiers résultats de l'appel sont utilisés :

```
-->deff('[a,b,c,d]=essai(x)', 'a=x ;b=2*x ;c=x^2 ;d=sqrt(x)')
-->essai(5)
 ans = 5.
-->[x,y]=essai(5)
 y = 10.
 x = 5.
-->[x,y,u,v]=essai(5)
 v = 2.236068
 u = 25.
 y = 10.
 x = 5.
```

Est-il raisonnable d'écrire des fonctions qui fournissent des résultats «secondaires» qui ne seront presque jamais utilisés ? SCILAB nous permet d'éviter cette perte de temps : il est possible de savoir si les résultats d'une fonction vont être, ou non, utilisés. La fonction `argn()` (qui doit être utilisée à l'intérieur d'une fonction définie) nous fournit le nombre de résultats effectivement utilisés, et le nombre de paramètres effectivement fournis lors d'un appel de cette fonction définie.

La fonction suivante est déclarée avec trois paramètres, et est supposée fournir trois résultats. Elle se contente, en fait, de donner comme résultat principal le nombre de résultats utilisés, et le nombre de paramètres fournis :

```
function [A,B,C]=ftst(X,Y,Z)
[u,v]=argn(0)
A=[u,v]
B=2
C=1
```

Voici quelques exemples d'appels :

```
-->x=ftst(1,2,3) ; x
x =! 1. 3. !
-->x=ftst(1) ; x
x =! 1. 1. !
-->[x,y,w]=ftst(1,2,3) ; x
x =! 3. 3. !
-->[x,y]=ftst() ; x
x =! 2. 0. !
```

En consultant `argn()`, une fonction peut donc déterminer si tel résultat secondaire est utilisé ou non, et éventuellement, éviter d'en calculer la valeur. On notera que la fonction `argn()`elle-même n'utilise pas son paramètre, et que n'importe quelle valeur convient pour ce dernier. Voici une réécriture de la fonction `sincos()` qui utilise cette approche :

```
function [u,v]=sincos(a)
k=argn()
u=sin(a)
if k==2 then
    disp("Je calcule le second résultat")
    v=cos(a)
end
```

Quelques appels :

```
-->k=sincos(0.135)
 k = 0.1345903
-->[u,v]=sincos(0.135)
 Je calcule le second résultat
 v = 0.9909013
 u = 0.1345903
```

5.7.5 Test d'existence de variables

La consultation de la valeur des variables externes offerte par SCILAB est complétée par la possibilité de savoir si une variable est définie ou non. La fonction `isdef()` prend comme premier paramètre le nom d'une variable, et indique si une valeur est associée ou non à ce nom :

```
-->isdef('toto')
ans = F
-->isdef('isdef')
ans = T
```

On utilisera typiquement cette construction pour donner une valeur à une variable lorsque aucune valeur externe n'est disponible, comme dans :

```
if ~ isdef("Colors") then
    Colors=[1 0 0      // Première couleur
            1 1 0 ] ;   // Seconde couleur
end ;
```

Une fonction similaire est remplie par l'opération `exists()`, qui rend, elle, un nombre, 0 pour *faux* et 1 pour *vrai*.

5.8 Exercices

5.8.1 Triangle de Pascal

Écrire une fonction, `Pascal(N)`, fournissant une matrice triangulaire inférieure contenant le triangle de *Pascal* d'ordre N. Ex :

```
-->Pascal(6)
ans =
! 1.  0.  0.  0.  0.  0. !
! 1.  1.  0.  0.  0.  0. !
! 1.  2.  1.  0.  0.  0. !
! 1.  3.  3.  1.  0.  0. !
! 1.  4.  6.  4.  1.  0. !
! 1.  5.  10. 10. 5.  1. !
```

5.8.2 Notations

Un enseignant dispose d'une matrice `Notes`, de taille $n \times p$, représentant les notes de ses n élèves lors de p interrogations. On suppose que tous les élèves étaient présents à chaque interrogation.

Comment calculer :
- la moyenne de chaque élève ;
- la moyenne de la classe pour chaque interrogation ;
- la moyenne totale de la classe.

Notre enseignant souhaite en outre obtenir pour chaque élève l'évaluation suivante : le nombre d'interrogations où l'élève a obtenu une note supérieure ou égale à la moyenne de la classe pour cette interrogation.

5.8.3 Nombres premiers

On se propose d'écrire une fonction `premiers(N)` qui génère la liste des nombres premiers inférieurs ou égaux à N. La fonction devra naturellement être la plus efficace possible pour de grandes valeurs de N.

5.8.4 Intégration de fonctions

On se propose d'écrire une fonction `MonteCarlo(f,a,b,N)` réalisant l'intégration d'une fonction $f(x)$ entre deux bornes, a et b, par la méthode de *Monte Carlo*. La surface à calculer, A, est incluse dans un rectangle de hauteur c et de largeur $b - a$. La surface de ce rectangle est donc $S = c \times (b - a)$. On génère aléatoirement N points répartis dans le rectangle, puis on compte le nombre de points M situés dans la surface A. L'intégrale cherchée est alors donnée par la formule : $S \times \frac{M}{N}$.

Application : on utilisera la fonction pour calculer les intégrales suivantes :

- $f1(x) = 4x^{\frac{1}{3}} + 3$, entre 1 et 8 (valeur attendue : 66) ;
- $f2(x) = \frac{1}{\sqrt{x}}$, entre 4 et 9 (valeur attendue : 2) ;
- $f3(x) = x \times \sqrt{1 - x^2}$, entre 0 et 1 (valeur attendue : 1/3) ;
- $f4(x) = \frac{x+6}{\sqrt{x+4}}$, entre 0 et 5 (valeur attendue : 50/3).

Chapitre 6

Bien programmer

6.1 Introduction

Il n'est pas inutile, arrivés à ce stade, de réfléchir aux divers aspects des programmes que nous avons écrits, et d'essayer d'en tirer quelques substantifiques leçons. Ce chapitre a donc pour objectif de faire réfléchir à divers aspects de la programmation, de proposer quelques conseils et techniques éprouvés, et surtout d'attirer votre attention sur le fait que la programmation est un *art* difficile... Vous n'y trouverez, hélas, pas de recettes magique, mais simplement de petits trucs et tours de mains sans prétention.

6.1.1 La meilleure façon de programmer...

Il y a plus de trente ans, l'informaticien *Jacques Arsac* énonçait : «La meilleur façon de programmer est encore d'utiliser un programme correct, écrit et prouvé par quelqu'un d'autre», ce qui est une autre manière de dire qu'il vaut mieux éviter de réinventer la roue, la votre pouvant être voilée ou se retrouver rapidement à plat.

6.1.2 Les dangers de la programmation

6.1.2.1 Il est toujours trop tard

Le premier danger qui guette le programmeur est celui du manque de temps. Lorsqu'il est face à son problème, avec des contraintes fortes de délais[1], il n'a plus le temps d'aller fouiller la littérature pour rechercher ce que dit *Knuth* dans [10], [11] ou encore [12] sur le sujet. Il ne peut utiliser que ce qu'il connaît déjà, approche qui, si l'on s'en réfère au proverbe bien connu «si le seul instrument dont vous disposez est un marteau, n'importe quoi se met à ressembler furieusement à un clou», la plupart du temps ne mène pas bien loin.

6.1.2.2 Quelques lois bien connues

Voici, présentées sous formes de boutades, un certain nombre de vérités profondes qu'il est utile de garder à l'esprit à l'occasion de toute activité de programmation :

[1] Un euphémisme pour dire qu'il est *déjà* en retard...

- Quand tout le reste a échoué, consultez le manuel[2].
- Si 90 % d'un programme prennent le temps T pour leur réalisation, 90 % de ce qu'il reste à faire prendront aussi le temps T.
- Les erreurs indétectables sont infiniment plus nombreuses que celles que vous songez à détecter.
- Les tests peuvent éventuellement montrer qu'un programme est incorrect ; jamais qu'il est correct.

Le lecteur intéressé trouvera moult autres judicieuses remarques et pertinents conseils en recherchant, par exemple, «*programming laws*» sur n'importe quel moteur de recherche internet...

6.2 Quel est le problème ?

Il s'agit probablement là de la difficulté principale. Spécifier proprement le problème sur le papier est la plus sûre voie vers une programmation efficace et correcte. Si vous ne savez pas résoudre le problème, vous ne saurez pas mieux comment transcrire votre solution en un programme SCILAB !

Il peut être utile, lors de la réflexion, d'associer au problème des exemples typiques de ses entrées et de des sorties attendues. Que le programme final soit cohérent avec ces sorties ne prouvera naturellement rien, mais la manipulation (manuelle) de ces données aura aidé au processus intellectuel de recherche de la solution.

Le problème peut être vague ou précis, la solution simple ou complexe ; par exemple :

- écrire des nombres (comme 63) sous forme littérale (soixante-trois) ;
- trouver les éléments uniques d'un vecteur.

Dans le premier cas, le problème est très bien précisé ; vous savez, a priori, ce que doit générer le programme pour chaque entrée. On imagine cependant que le programme va devoir traiter tout un ensemble de cas particuliers complexes, qu'il va falloir recenser avec soin. Dans le second cas, comment arriver efficacement au résultat est probablement plus vague dans votre esprit ; vous soupçonnez cependant que la solution, une fois trouvée, sera simple et courte à programmer.

Dans certains cas, il peut être nécessaire d'écrire un véritable «cahier des charges», expliquant votre point de vue particulier sur tout ce qui semble flou, décrivant les entrées et les sorties, etc.

6.3 Concevoir structures de données et algorithmes

6.4 Écrire des programmes efficaces

Tout au long des premiers chapitres, nous avons tenté de montrer que bien programmer impliquait, au minimum, deux conditions :

- bien connaître l'algorithmique ;
- bien connaître l'outil utilisé.

[2]Souvent présenté sour la forme RTFM, pour «Read The F*ck*ng Manual».

6.4.1 L'algorithmique

L'algorithmique sous-tend la moindre des activités de programmation. Il ne s'agit pas seulement de l'algorithmique de haut niveau, telle que l'on peut la trouver dans de remarquables pavés tels [6], [10], [11], [12] et bien d'autres, mais aussi de l'algorithmique élémentaire, dans laquelle il s'agit de gérer la variable d'un boucle, décrire un tableau, etc. Combien de programmes triviaux (quelques lignes) ne se révèlent-ils pas incorrects ?

Cependant, il n'est pas non plus inutile, lorsque l'on s'attaque à un problème nouveau, de chercher si celui-ci n'a pas déjà été abordé quelque part dans la littérature, et si des solutions performantes ne sont pas déjà connues, et publiées quelque part.

Illustrons ceci avec un problème bien connu, la suite de *Fibonacci*. Celle-ci est définie par la récurrence :

$$\begin{aligned} fib(0) &= 0 \\ fib(1) &= 1 \\ fib(n) &= fib(n-1) + fib(n-2) \end{aligned}$$

Le programme suivant est une implantation directe de la formulation ci-dessus :

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

L'analyse de ses performances les révèle plus que médiocres. L'algorithme est en $O(2^n)$, et le programme est donc extrêmement consommateur de temps calcul (l'une des ses rares utilisations est de tester les performances des machines et des langages...).

En fait, l'analyse du problème montre que la double récursion peut se simplifier, si l'on transmet à la fonction les deux valeurs $fib(n)$ et $fib(n-1)$ lorsqu'il s'agit de calculer $fib(n+1)$. La version suivante, toujours récursive, utilise 4 paramètres, l'ordre N demandé, un ordre n atteint au cours des calculs, et les valeurs correspondantes, U_n et U_{n-1} de la suite :

```
function Z=fib_aux(N,n,Un,Un_1)
if N>n then
    Z=fib_aux(N,n+1,Un+Un_1,Un) ;
else Z=Un ;
end
endfunction
```

Cette fonction peut être appelée par l'intermédiaire de la fonction suivante :

```
function Z=fib2(N)
U0=0 ; U1=1 ;
if N==0 then
    Z = U0 ;
```

```
else
    Z = fib_aux(N,1,U1,U0) ;
end
endfunction
```

Cette fonction teste le cas particulier $N = 0$ avant de faire appel à la récursion générale. Il serait également possible de décider qu'il existe un terme précédent $fib(0)$, dont la valeur serait $fib(-1) = fib(1) - fib(0)$, soit 1. L'écriture de `fib2()` se simplifie alors en :

```
function Z=fib2(N)
Z = fib_aux(N,0,0,1) ;
endfunction
```

Sous l'une de ces nouvelles formes, l'algorithme passe en $O(n)$.

On peut cependant aller plus avant, et constater que la récursion, dans la fonction `fib_aux()`, n'est qu'une boucle déguisée. La fonction peut se réécrire sous la forme :

```
function Z=fib3(N)
Un=0 ;
Un_1=1 ;
n=0 ;
while n<N do
    New = Un + Un_1 ;
    Un_1 = Un ;
    Un = New ;
    n = n+1 ;
end
Z=Un ;
```

Voici un tableau résumant les performances (en millisecondes) des trois fonctions ci-dessus, pour un paramètre égal à 30 :

n	fib(n)	fib2(n)	fib3(n)
30	108090	2.2302246	1.0144043

Cependant, même lorsque les temps obtenus peuvent être jugés «satisfaisants», on peut encore faire appel à la littérature, et trouver un algorithme plus performant que les précédents pour calculer $fib(n)$.

Le concept est le suivant : étant donné un couple d'entiers, (u, v), le produit de ce vecteur par la matrice $F = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$ fournit le résultat $(v, u+v)$, qui correspond à la fonction de récurrence de la suite de *Fibonacci*. En prenant $V = (0, 1)$, le produit $V \times F$ fournit $(1, 1)$, $V \times F \times F = (1, 2)$, etc :

```
-->F=[0 1 ;1 1]
 F =
! 0. 1. !
```

```
! 1. 1. !
-->V=[0 1]
 V =! 0. 1. !
-->V*F
 ans =! 1. 1. !
-->V*F*F
 ans =! 1. 2. !
-->V*F*F*F
 ans =! 2. 3. !
```

De manière générale, $V * F^n$ nous fournit les termes d'ordre n et $n+1$ de la suite. Or, élever un nombre (ou une matrice) à la puissance n peut se réaliser au moyen d'un algorithme en $O(\ln n)$. Nous avons donc ici l'outil théorique nous permettant de réaliser un algorithme calculant le terme d'ordre n de la suite de *Fibonacci* en $O(\ln n)$.

6.4.2 Lire et écrire

L'écriture de programmes comporte de nombreuses analogies avec l'écriture littéraire. Un fait est certain : un écrivain est d'abord un lecteur. Un programmeur devrait également consacrer du temps à lire des programmes corrects, modèles, et à s'en inspirer.

6.4.3 Connaître l'outil

Nous avons, à plusieurs reprises, présenté des programmes tels le suivant, qui calcule la somme des éléments d'un vecteur :

```
S=0 ;
for k=1 :count(V) do
    S=S+V(k) ;
end
```

Cette séquence d'instructions, qui est un peu l'archétype des boucles, est aisément transposable dans d'autres langages. Elle s'écrit de manière très similaire dans les langages C ou Pascal, et il est utile d'être familier avec cette construction, et de savoir la reconnaître à première vue.

SCILAB nous permet d'écrire ce calcul sous la forme suivante – approche qui lui est naturellement spécifique :

```
S=0 ;
for k=V do
    S=S+k ;
end
```

Il se trouve encore que ce besoin peut être satisfait avec l'opération de sommation `sum()`, sous la forme suivante :

```
S=sum(V) ;
```

Les tests montrent que la seconde solution est meilleure que la première (18 contre 28 millisecondes pour un vecteur de 1000 éléments), mais que la dernière solution, vectorielle, est largement préférable (0.07 millisecondes, soit 300 fois plus performante que les solutions précédentes).

6.4.4 Remarques sur l'efficacité

Il faut cependant être conscient que cette notion d'efficacité est toute relative, et dépend du référentiel dans lequel on se place.

On peut s'intéresser à l'efficacité de l'algorithme utilisé. Selon ces critères, la version 4 de *Fibonacci* est sans conteste la meilleure si l'on s'en réfère à son efficacité théorique.

On peut s'intéresser à l'efficacité de l'implantation. La version 4 de *Fibonacci* est sensiblement plus lente que la version 3 sur de petites valeurs du paramètre.

On peut enfin s'intéresser à l'adéquation entre l'effort de développement et le problème à résoudre. Si le critère est le coût du développement de la solution (l'exemple typique est celui du programme qui sera utilisé une seule fois), alors une solution qui ne prend que dix minutes de développement, avec un programme qui nécessitera dix minutes d'exécution, sera préférable à une solution qui réclame deux heures de développement, même si le programme résultant ne prend que deux secondes de temps de calcul... La version 1 de *Fibonacci* est alors préférable.

6.5 Forces et faiblesses du langage SCILAB

Comme tout langage de programmation, SCILAB a des points forts et des points faibles, dont il convient d'être conscient. Ces aspects sont en partie liés au fait que SCILAB se veut un clone de MATLAB, et que les qualités et défauts de l'un sont hérités de l'autre...

6.5.1 Points forts

Le langage SCILAB a plusieurs points forts.

SCILAB est un langage vectoriel. Il convient donc de privilégier systématiquement l'utilisation des primitives opérant sur les tableaux.

Au nombre des points forts, il faut mettre aussi la disponibilité de tout un ensemble de procédures et paquetages, largement validés et testés, qu'il convient d'utiliser chaque fois que c'est possible. Nombre de problèmes peuvent se résoudre par l'appel de quelques fonctions...

6.5.2 Points faibles

Les points faibles du langage se manifestent en revanche chaque fois qu'il est nécessaire d'écrire des procédures. Voici quelques reproches que l'on peut lui adresser :

6.5.2.1 Syntaxe et sémantique

La syntaxe du langage est inutilement complexe. De menus détails peuvent changer complètement la signification d'une construction. C'est ce qui se passe dans l'écriture de constantes vectorielles, où la moindre inattention, la plus minime faute de frappe, ne provoque pas une

erreur (ce qui ne serait pas dramatique), mais fournit une expression syntaxiquement correcte, qui malheureusement n'aura pas la sémantique désirée par le programmeur.

Le langage est *excessivement* laxiste. Beaucoup d'options par défaut, qui, dans l'opinion des concepteurs du langage, ont pour but de simplifier la vie du programmeur, sont susceptibles de se retourner contre celui-ci. Citons par exemple :

- le fait que les paramètres d'une fonction soient optionnels. Une fonction peut ainsi se retrouver appelée avec un nombre incorrect de paramètres, sans que ceci ne soit décelé. Notre fonction `add()` (c.f. § 3.5.1, page 63), qui ajoute ses deux paramètres :

  ```
  -->add(8,11)
   ans = 19.
  ```

  se retrouve fonctionner ainsi par accident, parce qu'il existe justement des variables `a` et `b` dans notre espace de travail :

  ```
  -->add()
   ans = 13.
  -->add(2)
   ans = 10.
  ```

- le fait qu'un tableau soit automatiquement créé ou redimensionné par indexation. Une erreur de calcul d'indice, par exemple, peut impliquer la création d'un nombre colossal de nouveaux éléments dans un vecteur. Une faute de frappe dans le nom d'une variable d'un programme va créer un nouveau tableau, qui recevra un élément modifié, alors que le tableau qui aurait dû être destinataire de cette valeur ne sera pas modifié. Ce type d'erreur est très difficile à découvrir.
- le fait qu'un indice non entier soit assimilé à l'entier dans une indexation... Un exemple :

  ```
  -->tab=[1 2 3 4 5]
   tab =! 1. 2. 3. 4. 5. !
  -->tab(5)
   ans = 5.
  -->s=0 ; for k=1 :171 do s=s+5/171 ; end
  -->s
   s = 5.
  -->tab(s)
   ans = 4.
  ```

  Dans cet exemple, des erreurs minimes de calculs font que la valeur finale de s est légèrement inférieure à 5, bien qu'il s'affiche sous la forme de ce nombre. Cependant, l'indexation, qui prend en compte la partie entière stricte, considère que la valeur effective de l'indice est 4.

6.5.2.2 Spécifications des primitives

Les spécifications des primitives sont en général *ad hoc*. Un exemple ? Voici deux variables, A et B, dont les valeurs diffèrent subtilement :

```
-->A = %pi+1E-15
 A = 3.1415927
```

```
-->B = %pi+1E-16
 B = 3.1415927
```

Or, une certaine fonction a la particularité de se comporter assez curieusement vis-à-vis de ces variables :

```
-->logspace(1,A,5)
 ans =! 10. 34.308218 117.70538 403.8262 1385.4557 !
-->logspace(1,B,5)
 ans =! 10. 7.4866489 5.6049912 4.1962601 3.1415927 !
```

Il se trouve en effet que *logspace(a,b,n)* fournit des éléments entre 10^a et 10^b, sauf si b a précisément la valeur π, auquel cas les éléments sont fournis entre 10^a et... π !

De manière plus générale, on reprochera à SCILAB de ne pas avoir de règles générales en ce qui concerne les domaines d'application, les natures, types, dimensions des résultats des fonctions fournies par le système. Ces incohérences sont décrites dans l'annexe dite «bétisier», qui pointe du doigt les principaux pièges du langage, et qu'il importe d'avoir survolé au moins une fois...

6.6 Quelques conseils de programmation

Une bonne présentation aide habituellement à mieux comprendre un programme, aussi bien lorsqu'on le lit que lorsqu'on l'écrit. Il existe des conseils assez universels, qui sont applicables à la quasi totalité des langages de programmation – donc SCILAB :

- *Écrire des programmes courts.*
  Un programme lisible, avec ses commentaires, ne doit pas dépasser une «page virtuelle», c'est-à-dire un écran, donc 20 à 30 lignes. Ceci veut donc dire qu'un gros programme doit avoir été décomposé en fonctions de petite taille, chacune accomplissant une tâche bien précise, et aisée à décrire en quelques lignes. Si une sous-fonction devient elle-même de taille démesurée, ça signifie qu'il est temps de la partager en quelques morceaux.
  Réciproquement, si l'on est amené à définir une fonction avec un nombre extravagant de paramètres, c'est probablement que cette fonction est trop grosse ou trop petite, et donc la décomposition de l'application mal effectuée.
- *Séparer les traitements des entrées-sorties.*
  Une application doit communiquer avec l'extérieur. Elle lit des données, imprime des résultats, trace des courbes, enregistre des informations sur des fichiers, etc. Il est crucial de décomposer cette application en modules distincts, dont le rôle sera :
  - lecture et validation des données ;
  - traitement des données ;
  - impression des résultats.

  Ces différents modules sont beaucoup plus simples à mettre au point séparément que lorsque toutes les fonctionnalités sont mélangées dans une même procédure. Cette approche favorise également la réutilisation de l'un ou l'autre de ces modules.
- *Utiliser des noms de fonctions et de variables significatifs.*
  Trop longs, ils sont ennuyeux et obscurcissent les programmes. Trop courts, ils sont trop vite oubliés. Trop cryptiques, ils provoqueront des erreurs et de mauvaises interprétations par le lecteur.

– *Utiliser des commentaires...*
Mais pas trop. Faire en sorte qu'ils soient réellement significatifs, mais non redondants. L'archétype du commentaire stupide :

```
k=k+1 ; // on ajoute 1 à k
```

– *Écrire pour les autres.*
Écrire afin d'être lu est un travail difficile. Idéalement, on devrait toujours se demander si ce que l'on écrit pourra être lu – et compris – par quelqu'un d'autre. Ou si, quelques semaines ou quelques mois plus tard, on sera soi-même capable de comprendre à nouveau ce que l'on avait écrit...

Comment présenter le texte d'un programme ? SCILAB est un langage à structure de blocs (les constructions syntaxiques sont imbriquées les unes dans les autres), et il est utile de faire ressortir cette imbrication en indentant (c'est à dire en décalant) les lignes qui sont à l'intérieur d'un *for*, d'un *while*, d'un *if*, etc.

Placer une instruction par ligne rend en général le programme plus lisible.

En revanche, des instructions courtes, fortement liées entre elles, auront intérêt à se trouver sur la même ligne, comme dans cet échange de deux variables :

```
tmp=p ; p=q ; q=tmp ;
```

Un exemple, la réécriture de la fonction de la fonction `bsort1()` (c.f. § 3.4.3.1) selon ces principes :

```
function Z = bsort1(V)
// Trier un vecteur de 0 et de 1 en
//    plaçant en tête les 0
Z = V ;
borneInf = 1 ;
borneSup = count(Z) ;
Elt = 0 ;
while borneInf < borneSup do
    if Z(borneInf) <> Elt then
        tmp = Z(borneInf) ; Z(borneInf) = Z(borneSup) ; Z(borneSup) = tmp ;
        borneSup = borneSup-1 ;
    else
        borneInf = borneInf+1 ;
    end
end
```

Mes conventions (implicites) consistent à dire que le résultat d'une fonction a pour nom `Z`, que le paramètre s'appelle `V` car on attend un vecteur ; les autres variables portent des noms liés à leur fonction. En résumé, tout ce qui peut rendre le code plus lisible est légitime.

6.7 Sécurisation des programmes

On l'a dit, un programme devrait idéalement se composer d'un ensemble de boîtes noires logicielles, sous forme de procédures ou d'ensemble de procédures, chacune définie avec soin. Ceci veut dire que :

- la finalité de la procédure est parfaitement décrite, de manière simple et compréhensible ;
- la procédure utilise le moins possible de paramètres, fournit le moins possible de résultats, utilise le moins possible de variables externes ;
- la procédure est correctement *blindée*. Elle effectue la vérification de ses entrées, et n'accepte pas d'entrée hors spécifications (par exemple, si une procédure est supposée s'appliquer à un vecteur de deux éléments, elle ne doit pas accepter un vecteur de trois éléments dont elle n'utiliserait que les deux premiers).

6.7.1 Vérification d'assertion

Il peut être utile dans certaines fonctions de tester certaines conditions particulières, et de signaler une erreur si ces conditions ne sont pas remplies. La fonction `assert` ci-dessous admet deux paramètres. Le premier est une expression logique, qui sera typiquement un test d'existence, de compatibilité, etc ; le second est un message d'erreur, qui sera imprimé si le test n'est pas satisfait, tout en interrompant l'exécution du programme.

Exemple d'utilisation :

```
-->assert(n==p, 'Dimensions des matrices non compatibles')
-->assert(2>5, 'Dimensions des matrices non compatibles')
    !--error 655 Assert :Dimensions des matrices non compatibles
at line 3 of function assert
called by : assert(2>5, 'Dimensions des matrices non compatibles')
```

Voici le code du programme :

```
function assert(condition, message)
if ~ condition then
    error("Assert :"+message, 655) ;
end
```

La procédure fait appel à la fonction `error`, qui positionne un numéro d'erreur (ici 655), imprime un message (son premier paramètre), et interrompt l'exécution. Naturellement, cette procédure `assert` n'est pas parfaite. Elle suffit dans nos petits exercices à détecter les erreurs de nos algorithmes.

Idéalement, on l'utilisera avec la fonction `check` suivante, identique dans l'esprit :

```
function check(condition, message)
if ~ condition then
    error("Check :"+message, 654) ;
end
```

Pourquoi deux fonctions ? On peut convenir que `check` nous servira à valider les entrées d'une procédure (les erreurs éventuelles sont donc dues au *programme appelant*), tandis que `assert` nous permettra de valider des assertions faites sur le fonctionnement de la *procédure elle-même*. L'échec d'une assertion est alors dû à une erreur de programmation, ou d'algorithmique, au sein de cette procédure.

6.7.2 Sous-systèmes

Il peut sembler paradoxal de prôner d'un côté l'écriture de *petites* fonctions, et d'un autre la réalisation de fonctions qui vérifient *minutieusement* leurs entrées et leur environnement avant de passer au vif du sujet, c'est-à-dire l'algorithme qu'elles doivent réaliser. On peut cependant admettre qu'une *sous-application*, ou un *sous-système*, consiste en une ou plusieurs fonctions principales, «visibles» de l'extérieur, et un certain nombre de fonctions auxiliaires, qui n'ont pas à être utilisées à l'extérieur de ce sous-système.

Dans ces conditions, les fonctions principales doivent effectivement être strictes sur les paramètres qu'elles acceptent. On peut imaginer au contraire que les fonctions auxiliaires sont utilisées dans un contexte «propre», ne sont appelées qu'avec des paramètres adéquats, et n'ont donc pas à vérifier ceux-ci.

SCILAB ne permet pas de «cacher» les fonctions auxiliaires. On peut convenir que les noms de ces fonctions auxiliaires débutent pas un préfixe commun, caractéristique du sous-système. On évite ainsi des conflits de noms éventuels avec d'autres fonctions appartenant à d'autres sous-systèmes.

Dans la pratique, comment organiser un «sous-système» ?

Une première solution consiste à placer fonctions principales et fonctions auxiliaires dans un fichier unique, ce qui fait que l'ensemble des fonctions est chargé en une fois.

Une autre solution consiste à placer tous les fichiers définissant les différentes fonctions de l'application dans un même répertoire, et à charger l'ensemble des fonctions du répertoire par la commande `getd()`.

Il est enfin possible de développer une bibliothèque, qui rassemblera les fonctions (consulter pour ce faire les commandes `lib()` et `genlib()`). La bibliothèque est alors définie par une simple référence dans la zone de travail, et les fonctions sont chargées automatiquement la première fois qu'elles sont utilisées.

6.8 Notes sur la mise au point des programmes

Nous avons présenté ci-dessus la technique des vérifications et assertions, instructions qui peuvent rester à demeure dans un programme, même une fois terminée la phase de mise au point. Voici quelques mots sur l'utilisation d'arrêts et de trace dans les cas difficiles de mise au point. Insistons malgré tout à nouveau sur l'idée qu'il vaut mieux essayer de faire bien dès le départ, plutôt que de passer son temps à bricoler continuellement un programme mal conçu...

6.8.1 Arrêts

Nous avions signalé l'existence de l'instruction `pause()`, qui, placée à l'intérieur d'une fonction, permet de suspendre l'exécution de celle-ci. Il est également possible de positionner un *point d'arrêt* (ou *break-point*) associé à une ligne spécifique d'une fonction, au moyen de l'opération `setbpt()`.

Les autres fonctions associées aux points d'arrêts sont `dispbpt()`, qui imprime la liste des points d'arrêt définis, et `delpbt()`, qui permet de retirer les points d'arrêts d'une fonction.

6.8.2 Trace

Une autre technique de mise au point, effective, est la pause de *traces*, c'est-à-dire d'ordres d'impression des valeurs de certaines variables locales, afin de suivre l'évolution d'un algorithme. Il est bien de faire en sorte que ces traces soient conditionnées, par exemple par la consultation d'un indicateur global :

```
if Traces then disp("toto ="+string(toto)) ; end ;
```

La variable globale `Traces` peut être positionnée interactivement après un arrêt en un certain point du programme.

6.9 Exercices

Les exercices proposés dans ce chapitre sont un peu inhabituels. Ils ont pour but de vous faire exercer votre perspicacité pour résoudre des problèmes en apparence anodins...

6.9.1 Gags

Expliquez les résultats suivants :

1. `-->count Dracula`
   `ans = 1.`
2. `-->help ;`
   affiche l'aide pour «help» – et non pour le point-virgule.

Étant donnée une matrice numérique `M`, l'expression `zeros(M)` permet d'obtenir une nouvelle matrice, de même taille, mais dont tous les éléments sont à zéro. Peut-on obtenir le même résultat, avec une expression plus courte ? Comparez l'efficacité des diverses solutions.

6.9.2 L'indice erroné

La fonction `find()` a pour but de découvrir dans un tableau numérique les indices des valeurs non nulles, et dans un tableau booléen les indices des valeurs vraies.

```
-->A=[F  F  F  F  T  F]
 A =!  F  F  F  F  T  F!
-->B=[F  F  T  F  T  T]
```

```
 B =! F F T F T T !
-->find([A,B])
 ans =! 5. 9. 11. 12. !
-->find([A ;B])
 ans =! 6. 9. 10. 12. !
```

Comment expliquez-vous les différences entre ces deux derniers résultats ?

6.9.3 La meilleure expression

Quelle est, pour chacun des problèmes décrits ci-dessous, la «meilleure» expression (s'il en existe une), soit en terme d'espace en mémoire, soit en terme de temps d'exécution. Réfléchissez avant de donner votre interprétation, en classant éventuellement les solutions de la meilleure à la plus mauvaise . Tester ensuite au moyen de la fonction `chrono`. Proposez enfin de meilleures solutions.

– création d'un vecteur particulier, les n premiers nombres impairs :

```
V=[] ;for k=1 :10000 do V(k)=2*k-1 ; end ;
V=zeros(1,10000) ;for k=1 :10000 do V(k)=2*k-1 ; end ;
V=[] ;for k=10000 :-1 :1 do V(k)=2*k-1 ; end ;
V=zeros(1,10000) ;for k=10000 :-1 :1 do V(k)=2*k-1 ; end ;
```

En déduire une solution de l'exercice proposé au § 5.2.1.

– somme des éléments positifs d'un vecteur

```
sum(max(V,0)) ;
sum(V(V>0)) ;
```

6.9.4 Diagonale

Soit une matrice carrée M. On se propose d'écrire toutes les expressions qui fournissent la matrice diagonale D dont la diagonale est égale à celle de M. Ex :

```
-->M
M =
! 0.2113249 0.6653811 0.8782165 0.7263507 !
! 0.7560439 0.6283918 0.0683740 0.1985144 !
! 0.0002211 0.8497452 0.5608486 0.5442573 !
! 0.3303271 0.6857310 0.6623569 0.2320748 !
-->D
D =
! 0.2113249 0.         0.         0.        !
! 0.         0.6283918 0.         0.        !
! 0.         0.         0.5608486 0.        !
! 0.         0.         0.         0.2320748 !
```

6.9.5 Les tours de Hanoi

Nous allons proposer un exemple bien connu, celui des tours de Hanoi. Nous nous inspirerons, pour la présentation du problème, de l'excellent ouvrage de Robert Cori et Jean-Jacques Lévy ([5]).

Le problème des tours de Hanoi consiste à déplacer une pile de disques d'un emplacement à un autre. Les contraintes à respecter sont les suivantes :

- les disques sont initialement empilés, du plus grand (en dessous), au plus petit (au dessus) ;
- on ne peut déplacer qu'un disque à la fois ;
- on dispose d'un emplacement auxiliaire, sur lequel on peut empiler un nombre quelconque de disques ;
- on ne peut poser un disque que sur l'un des emplacements, ou sur un disque plus grand que lui ;
- les disques doivent naturellement être empilés, à l'emplacement d'arrivée, du plus grand au plus petit.

Une légende (en réalité inventée par le mathématicien français E. Lucas en 1883) dit que les bonzes passaient leur vie à Hanoi à résoudre ce problème pour $n = 64$, la fin du monde étant supposée se produire une fois le problème résolu.

Le raisonnement par récurrence permet de résoudre assez simplement le problème ;

- celui-ci est trivial pour $n = 1$;
- imaginons le problème résolu jusqu'à l'ordre $n - 1$. Pour transférer n disques de l'emplacement A à l'emplacement B, il suffit de transférer $n - 1$ disques de A vers C, le dernier disque, le plus grand, de A vers B, puis les $n - 1$ disques de B vers C. Cette manipulation respecte les diverses contraintes du problème.

Nous désignons les emplacements par des numéros de 1 à 3, ce qui, étant donné deux emplacements A et B, permet d'obtenir le numéro du troisième emplacement par $6 - A - B$. La fonction suivante nous donne une première version de la chose :

```
function han(N,A,B)
if N>0 then
    C=6-A-B
    han(N-1,A,C)
    disp(string(A)+" -> "+string(B)) ;
    han(N-1,C,B)
end
```

Faire passer 3 disques du piquet 1 au piquet 2 s'écrit :

```
-->han(3,1,2)
 1 -> 2
 1 -> 3
 2 -> 3
 1 -> 2
 3 -> 1
 3 -> 2
 1 -> 2
```

Un message tel que `3 -> 1` signifie «faire passer le disque du sommet du piquet 3 vers le piquet 1». Ce programme est naturellement remarquable de sobriété, tant dans sa programmation que dans ses sorties :-).

Facile l'on désire afficher, pour mieux s'y retrouver, le numéro du disque déplacé. Comment effectuer ces modifications dans notre programme `han()` ci-dessus ?

Moins simple le travail de notre fonction `han()` a pour but de faire passer les disques d'une *configuration hanoi*[3] particulière (tous les disques sont sur un piquet donné) à une autre configuration hanoi particulière (tous les disques sont sur un autre piquet). Comment modifier notre programme pour pouvoir passer d'une *configuration hanoi* quelconque à une autre *configuration hanoi* quelconque ? Comment s'assurer que l'ensemble des déplacements obtenus est optimal ?

[3] On désignera sous ce terme une configuration des disques dans laquelle ceux-ci sont répartis sur les divers piquets, en respectant la règle de base, à savoir qu'un disque quelconque est toujours posé sur le sol ou sur un autre disque plus grand que lui.

Annexe A

Quelques utilitaires

A.1 Fichier de démarrage

Le fichier de démarrage `.scilab`, placé dans le répertoire d'accueil de l'utilisateur, est automatiquement exécuté chaque fois que le logiciel est lancé. La configuration suivante permet de s'affranchir de quelques-unes des différences entre MATLAB et SCILAB, et fournit les fonctions utilisées dans ce cours.

```
// Fichier de démarrage pour Scilab
// Définition de variables prédéfinies
pi = %pi
i = %i
eps = %eps
nan = %nan
inf = %inf
t = %t
f = %f
T = %t
F = %f
// Création de vecteur et matrices
deff('r=vec(x)','r=zeros(1,x)')
deff('r=mat(x,y)','r=zeros(x,y)')
```

A.2 Quelques Fonctions utiles

A.2.1 Compte d'éléments

Systématiquement utilisée dans ce poly, la fonction `count(A)` nous fournit le nombre d'éléments de l'objet `A` :

```
function Z=count(A)
Z=prod(size(A))
```

A.2.2 Chronométrage d'une instruction

La procédure suivante accepte en paramètre une chaîne de caractères, représentant une instruction exécutable, et chronomètre le temps nécessaire pour exécuter cette instruction. Le résultat est affiché en millisecondes. Ce résultat est la durée d'une exécution individuelle. La fonction teste des facteurs de répétitions croissants afin d'obtenir une précision suffisante pour l'évaluation du temps d'exécution.

Voici le code source de cette fonction :

```
function chrono(str)
// Répéter une expression un nombre suffisant de fois
// La fonction fournit le temps d'exécution individuel
repeat = 1 ;
cycle=%t ;
while cycle do
  r=timer() ;
  c=repeat ;
  while c>0 do
    execstr(' ;') ;
    c = c-1 ;
  end
  r=timer() ;
  c=repeat ;
  while c>0 do
    execstr(str) ;
    c = c-1 ;
  end
  w=timer() ;
  if w>10 | repeat >= 1000000 then
    cycle=%t ;
    r=1000*abs(w-r)/repeat ;
    disp(string(r)+' ms.') ;
  end
  repeat=repeat*2 ;
end
```

Table des figures

Bibliographie

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Mass., 1985.

[2] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure et Interprétation des Programmes Informatiques*. InterEditions, Paris, France, 1989.

[3] Alfred Aho, John Hopcroft, and Jeffrey Ullman. *Structures de données et algorithmes*. InterEditions, Paris, France, octobre 1987.

[4] Jon Bentley. *Programming Perls*. Mark S. Dalton, U.S.A., april 1986.

[5] Robert Cori and Jean-Jacques Lévy. *Algorithmes et Programmation*. École Polytechnique, 1994.

[6] Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction à l'algorithmique*. Dunod, Paris, France, août 1997.

[7] Claude Gomez. *Engineering and Scientific Computing with Scilab*. Birkhäuser, 1999.

[8] Scilab Group. Introduction to Scilab. Technical report, INRIA, 1998.

[9] Michel Habib. Algorithmique et programmation. Support de cours.

[10] Donald E. Knuth. *Fundamental Algorithms*. Addison-Wesley, U.S.A., 1968.

[11] Donald E. Knuth. *Semunumerical Algorithms*. Addison-Wesley, U.S.A., 1969.

[12] Donald E. Knuth. *Sorting and Searching*. Addison-Wesley, U.S.A., 1973.

[13] Donald E. Knuth. Big omicron and big omega and big theta. *SIGACT News*, April-June 1976.

[14] Jean-Thierry Lapresté. *Introduction à Matlab*. Ellipses Édition, 1999.

[15] Charles F. Van Loan. *Introduction to Scientific Computing*. Prentice-Hall Inc, 2000.

[16] Bruno Pinçon. *Une introduction à Scilab*. Institut Elie Cartan Nancy, 1999.

[17] Christian Rolland. *LaTeX 2ε, guide pratique*. Addison-Wesley, France, Juin 1995.

[18] Kermit Sigmon. *Matlab, Aide-Mémoire*. Springer-Verlag France, 1999.

[19] LyX Team. Implémentation de LyX. http ://www.lyx.org/.

[20] Matlab team. *Using Matlab*. The Math Works Inc, 1998.

[21] Lydia E. van Dijk and Christoph L. Spiel. *Scilab Bag Of Tricks, The Scilab-2.5 Infrequently Asked Questions*. Free Software Fundation, 1999.

Index