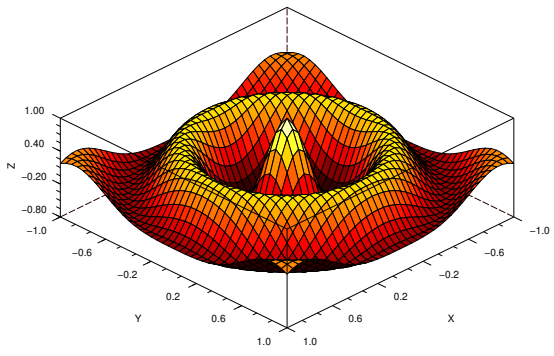
Informatique

Saint Joseph La Salle LORIENT

Marc Beutier
obelix56@free.fr
http://obelix56.free.fr

cpge TSI

Établissement St Joseph - LaSalle Lorient



book_20-06_scilab_eleve 30 juin 2020

Python & Scilab

Version Élève

CPGE TSI Lorient

Ce fascicule s'adresse avant tout à vous, étudiants de première année de CPGE. J'espère qu'il pourra vous servir de base pour l'année et pourquoi pas pour les années suivantes ?
Il sera mis à jour sur

http://obelix56.free.fr

Vous trouverez également sur ce site les dossiers *eleve* dans lequels se trouvent tous les programmes écrits ici en *python* (*.py*) et *scilab* (*.sci* et *.sce*).
Les programmes proposés durant les activités et ceux demandés en exercices ne sont pas disponibles, pour le bon déroulement des séances ...
La progression de l'année sera calquée sur ce fascicule et comme nous ne trouvons le temps de tout faire, vous pourrez combler vos lacunes le soir, avant de vous endormir ...

Le fascicule existe rassemble le programme des deux années en algorithmique, sur *Scilab* , sur *Python* et sur le traitement d'images.

Je ne peux remercier toutes les personnes qui m'ont aidé à rédiger ce fascicule. Entre les collègues, les auteurs d'ouvrages de référence, les tutoriels, FAQ et autres forums, je tiens particulièrement à remercier Arnaud Coatanhay et Christophe Osswald, enseignants - chercheurs à l'ENSTA Bretagne, pour leur disponibilité et la qualité de leur formation en *Python* et Philippe Roux, Jean-Paul Chehab, Jérôme Briot et Michael Baudin pour le partage de leurs ouvrages en format LaTeXconcernant *Scilab*.
"Une introduction à Python 3" écrit par Bob Cordeau et Laurent Pointal a aussi certainement inspiré mon travail. Leurs sources LaTeXet *Python* sont disponibles sur leur site. On peut trouver cela ici :

http://perso.limsi.fr/pointal/python:courspython3

Pour ce qui est des forums, celui que j'utilise régulièrement est :

http://www.developpez.net/forums/f96/autres-langages/python-zope/

En cas de réclamation, veuillez me contacter à l'adresse indiquée sur la première page.

Table des matières

Partie 1

Scilab année 1

Table des matières

1

Introduction

1.1 Vue d'ensemble de *Scilab*

Scilab est un langage de programmation associé à une riche collection d'algorithmes numériques couvrant de nombreux aspects des problèmes de calcul scientifique.

Du point de vue logiciel, *Scilab* est un langage interprété. Ceci accélère généralement le processus de développement, parce que l'utilisateur accède directement à un langage de haut niveau, avec un riche ensemble de fonctionnalités offertes par la bibliothèque.
Le langage *Scilab* est destiné à être étendu afin que des types de données " utilisateurs " puissent être définis par d'éventuelles opérations de surcharge. Les utilisateurs de *Scilab* peuvent développer leurs propres modules afin de résoudre leurs problèmes particuliers. Le langage *Scilab* peut compiler et lier dynamiquement d'autres langages tels que *Fortran* et *C* : de cette façon, des bibliothèques externes peuvent être utilisées comme si elles faisaient partie des fonctionnalités intégrées de *Scilab*.
Scilab s'interface également avec *LabVIEW*, une plate-forme et un environnement de développement pour le langage de programmation visuel de National Instruments.

Du point de vue de la licence, *Scilab* est un logiciel gratuit et open source, sous licence *Cecill*. Le logiciel est distribué avec le code source, de telle sorte que l'utilisateur dispose d'un accès aux aspects les plus internes de *Scilab*. La plupart du temps, l'utilisateur télécharge et installe une version binaire de *Scilab*, car le consortium *Scilab* fournit les versions exécutables pour *Windows*, *Linux* et *Mac OS*. L'aide en ligne est disponible dans de nombreuses langues.

Scilab est donc un logiciel libre dont vous pouvez vous procurer la dernière version sur le site de l'INRIA (*Institut National de la Recherche en Informatique et en Automatique*) :
https://www.scilab.org/fr/download/5.4.1

Vous y trouverez les fichiers nécessaires pour l'installer sur n'importe quelle plate-forme.
Techniquement, *Scilab* est un interpréteur de commandes : il se présente donc comme une calculatrice ou un shell. Les commandes, tapées à l'invite des commandes ou lancées à partir d'un script, sont traduites dans un autre langage (*Fortran* ou *C*), compilées, puis exécutées par le noyau du système d'exploitation, le résultat (ou l'erreur) est récupéré par *Scilab* et affiché dans la console.

Du point de vue scientifique, *Scilab* est livré avec de nombreuses fonctionnalités. Au tout début de *Scilab*, les fonctionnalités se sont concentrées sur l'algèbre linéaire. Mais, rapidement, leur nombre s'est élargi pour couvrir de nombreux domaines de l'informatique scientifique. Voici une courte liste de ses capacités :

- algèbre linéaire, matrices creuses,
- polynômes et fractions rationnelles,
- interpolation, approximation,

- optimisation linéaire, quadratique et non linéaire,
- solveur d'équations différentielles ordinaires et solveur d'équations différentielles algébriques,
- commande classique et robuste, optimisation par inégalité matricielle linéaire,
- optimisation différentiable et non différentiable,
- traitement du signal,
- statistiques.

Scilab offre de nombreuses fonctionnalités graphiques, y compris un ensemble de fonctions de traçage, qui créent des tracés $2D$ et $3D$ ainsi que des interfaces utilisateur. L'environnement *Xcos*, que vous utiliserez en Sciences Industrielles, fournit un modeleur et un simulateur hybride de systèmes dynamiques. Vous pourrez trouver un tutoriel pour débuter en *Xcos* ici :
www.scilab.org/fr/content/download/1107/10091/file/Xcos_debutant.pdf
Un autre tutoriel plus complet pourra être téléchargé ici :
www.scilab.org/fr/content/download/1017/9485/file/livret_Xcos.pdf
Plus généralement, vous pourrez trouver un tas de ressources ici :
http://wiki.scilab.org/Tutorials
Voici sur la figure 1.1 un aperçu de la fenêtre qui apparaît au démarrage de *Xcos* :

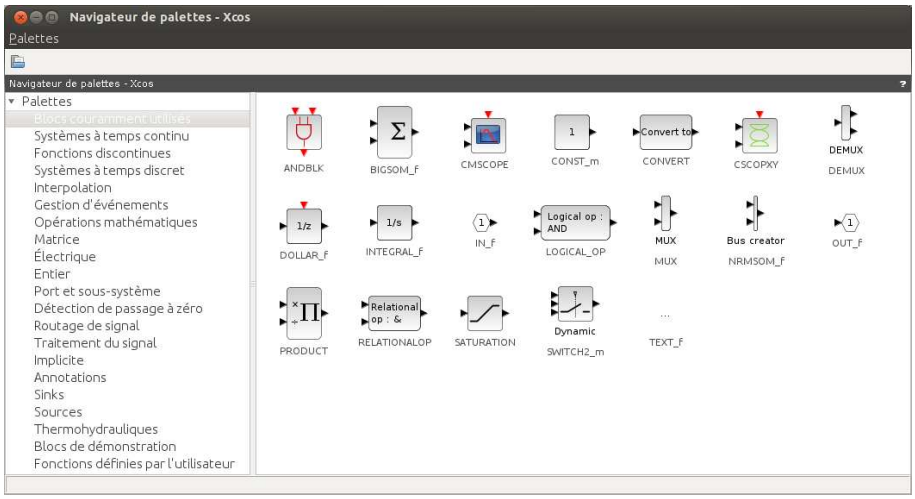


FIGURE 1.1 – Xcos

1.2 Installation

1.2.1 Sous *Windows*

Scilab est distribué sous forme binaire pour *Windows* et un programme d'installation est fourni pour que l'installation soit vraiment facile.
Si on choisit la langue française lors de l'installation, alors *Scilab* est installé en Français.
L'environnement de travail *Scilab* est présenté dans la figure 1.2.

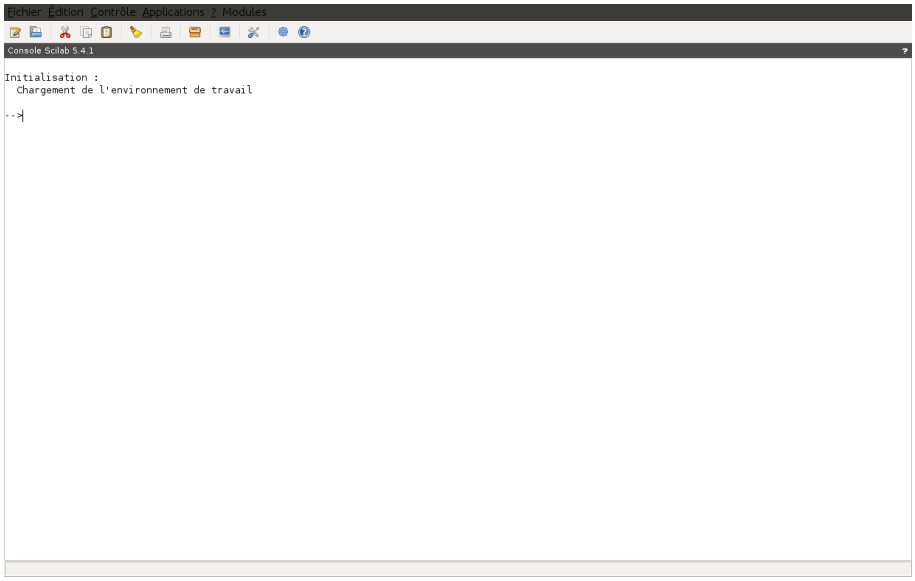


FIGURE 1.2 – Environnement de travail

1.2.2 Sous *GNU/Linux*

Scilab est distribué avec le système de paquets disponibles avec les distributions *GNU/Linux* basées sur *Debian* (par exemple *Ubuntu*), donc à partir de la logithèque. Cette méthode d'installation est extrêmement simple et efficace. Néanmoins, il y a un petit inconvénient : la version de *Scilab* empaquetée pour votre distribution *Linux* peut ne pas être à jour.
Scilab est alors installé dans *Dossier personnel/.Scilab/scilab-5.x.x*.

1.2.3 Sous *Mac OS*

Sous *Mac OS*, les versions binaires sont disponibles sur le site de *Scilab* sous la forme d'un fichier `.dmg`. Ce binaire fonctionne pour les versions *Mac OS* à partir de la version 10.5. Il utilise le programme d'installation de *Mac OS*, ce qui fournit un processus d'installation classique.

1.3 Prise en main de *Scilab*

Lorsque vous démarrez *scilab*, la fenêtre principale s'ouvre comme sur la figure 1.2. Un curseur apparaît juste après l'invite des commandes (- - >) : c'est à cet endroit que vous pourrez lancer les lignes de commandes qui seront exécutées séquentiellement, de la même manière qu'avec une calculatrice. Pour pouvoir utiliser la console, il faut comprendre les quelques règles suivantes :

- chaque commande *Scilab* doit se terminer par :
  - un retour à la ligne (↩) : ce qui nous limite à une commande par ligne,
  - une virgule (,) : ce qui permet de mettre plusieurs commandes par ligne,
  - un point virgule (;) : ce qui permet aussi d'exécuter plusieurs commandes par ligne en bloquant l'affichage du résultat précédant le point virgule. Le point virgule est très utile pour masquer le résultat d'un calcul intermédiaire.
  - 3 points de suspension (...) : ceci permet de prolonger une ligne en allant à la ligne. Cela est utile lorsqu'une commande est trop longue pour tenir en une seule ligne, par exemple quand on écrit une expression arithmétique kilométrique.
    Exemple :

```
--> x=10*(a*sin(b)+...
--> 3*b\^{}2);
```

- On peut copier/coller des instructions avec la souris (mais pas modifier une ligne de commande déjà exécutée)
- On peut rappeler les commandes précédemment exécutées en utilisant les touches du curseur ↑ et ↓.
- dans une ligne tout ce qui suit un double slash (`//`) est ignoré (ce qui permet d'insérer du commentaire).
- *Scilab* possède une aide en ligne accessible depuis la barre de menus, dans l'onglet ? : voir figure 1.3.

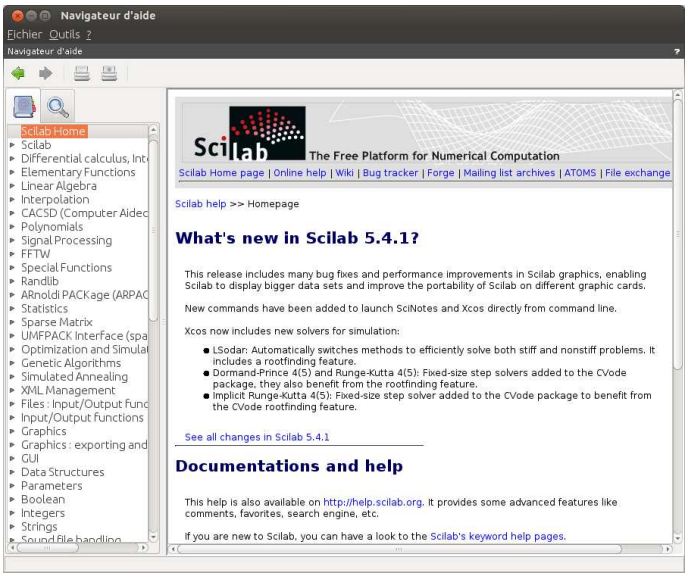


FIGURE 1.3 – L'aide en ligne de *Scilab*

Vous pouvez aussi accéder à l'aide en ligne de *Scilab* directement depuis la console. Pour cela, il y a deux commandes différentes suivant le type de recherche qu'on veut faire dans l'aide en ligne :

- `help commande` si vous voulez connaître ce que fait une *commande* dont vous connaissez déjà le nom,
- `apropos mot_clé` si vous cherchez des commandes ayant rapport avec un thème décrit par un `mot_clé`.

par exemple :

```
-->help sin
-->apropos sinus
```

La première commande va trouver la page d'aide de la fonction sinus (`sin` en *Scilab*) alors que la deuxième va renvoyer la liste des commandes ayant un rapport avec la fonction sinus (en fait, la liste des fonctions trigonométriques comme sur la figure 1.3).

Pour le descriptif des commandes *Scilab]* vous pouvez également consulter le site :
http://www.iut-fbleau.fr/math/alglin/telechargement/manual_scilab-5.0.1_fr_FR/

Vous pouvez également consulter les démos :

FIGURE 1.4 – Démos

⚠ Une dernière chose importante qu'il faut comprendre pour bien travailler avec *Scilab* c'est la notion de *répertoire courant*. On a souvent besoin avec *Scilab* d'importer des informations, ou au contraire d'en exporter, et ceci à l'aide de fichiers. Lorsque c'est le cas, *Scilab* va chercher à effectuer ces opérations avec les fichiers présents dans son répertoire courant.

Au démarrage de *Scilab*, ce répertoire est le répertoire depuis lequel *Scilab* a été lancé. Il existe plusieurs fonctions dédiées à la manipulation des répertoires courants :

À partir du menu *fichier* ci-dessous :

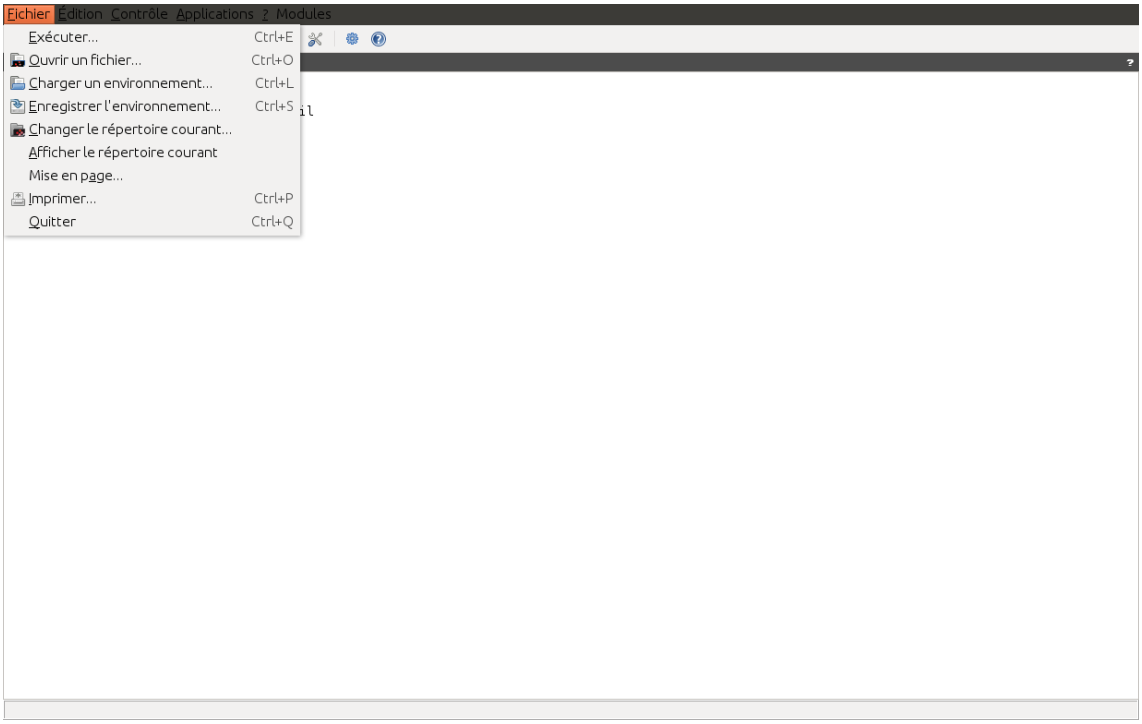


FIGURE 1.5 – Menu fichier

On accède au répertoire courant : cf figure 1.6.

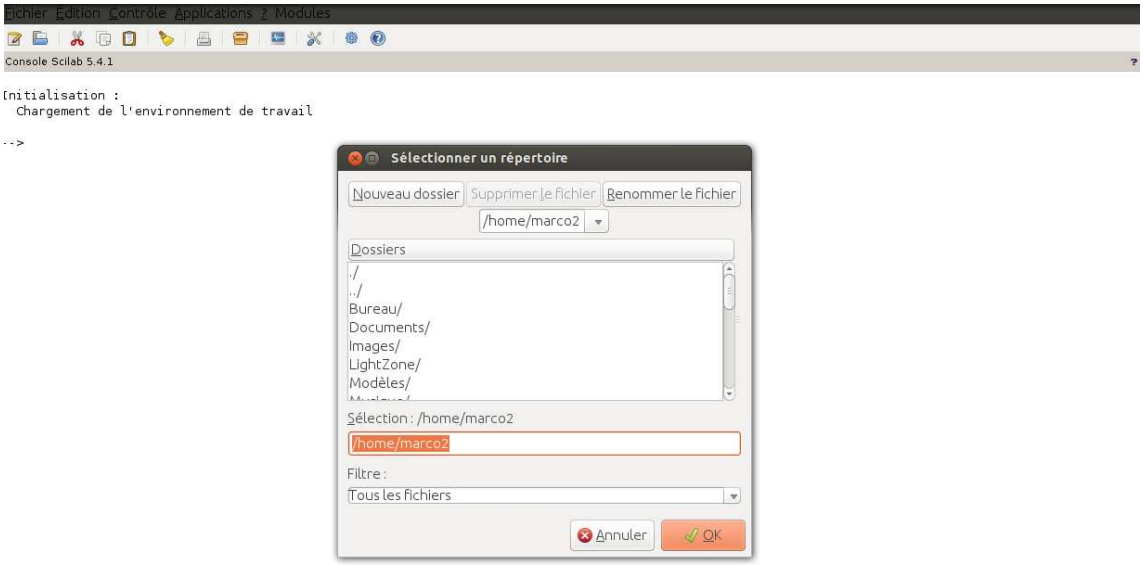


FIGURE 1.6 – Répertoire courant

1.4 Listes de diffusion, forums, . . .

La liste de diffusion users@lists.scilab.org est conçue pour toutes les questions d'utilisation de *Scilab* (les échanges ont lieu en anglais).
Pour souscrire à cette liste de diffusion, il suffit d'envoyer un e-mail à l'adresse :
users-subscribe@lists.scilab.org.
La liste users-fr@lists.scilab.org est destinée aux utilisateurs francophones de Scilab.
La liste de diffusion dev@lists.scilab.org se concentre sur le développement de *Scilab*, que ce soit le développement du noyau *Scilab* ou de modules complexes qui interagissent fortement avec le noyau *Scilab*.
Ces listes de diffusion sont archivées à l'adresse : http://mailinglists.scilab.org.

Par conséquent, avant de poser une question, les utilisateurs peuvent regarder dans l'archive si la même question ou le sujet a déjà obtenu une réponse.
D'autres listes de diffusions sont également disponibles ici :
http://www.scilab.org/fr/development/ml.

Un forum *Scilab* en français est disponible sur *Developpez.com* à cette adresse :
http://www.developpez.net/forums/f1715/environnements-developpement/autres-edi/scilab.

D'autres forums de discussion existent sur *Scilab*, en particulier le forum animé par *Equalis* :
www.equalis.com.

1.5 Ancrage des fenêtres graphiques

L'interface graphique de la version *Scilab 5* a été mise à jour de sorte que de nombreux composants sont désormais basés sur Java. Cela a un certain nombre d'avantages, notamment la possibilité de gérer l'ancrage des fenêtres.
Le système d'ancrage [1] des fenêtres graphiques utilise *FlexDock*, un projet open source fournissant un système d'ancrage *Swing*.

1. en anglais : *docking*

Le système *FlexDock* permet de faire glisser et de déposer l'éditeur dans la console, pour avoir, finalement, une seule fenêtre, avec plusieurs sous-fenêtres.
Toutes les fenêtres *Scilab* peuvent être ancrées, y compris la console, l'éditeur, le navigateur de variables, l'historique des commandes, l'aide et les fenêtres de tracé.
Afin d'ancrer une fenêtre à une autre fenêtre, il faut faire glisser la fenêtre source et la déposer dans la fenêtre cible.

1.6 Utiliser exec

Lorsque plusieurs commandes doivent être exécutées, il peut être plus commode d'écrire ces déclarations dans un fichier avec l'éditeur de *Scilab*. Pour exécuter les commandes situées dans un tel fichier, la fonction `exec` peut être utilisée, suivie par le nom du script.
Ce fichier a généralement l'extension `.sce` ou `.sci`, en fonction de son contenu :

- les fichiers ayant l'extension `.sci` contiennent des fonctions *Scilab* : exécuter le fichier charge alors les fonctions dans *Scilab* (mais n'exécute pas les fonctions)
- les fichiers ayant l'extension *.sce* contiennent à la fois des fonctions *Scilab* et des instructions exécutables.

Exécuter un fichier *.sce* a généralement pour effet de calculer plusieurs variables et d'afficher les résultats dans la console, de créer des tracés 2D, de lire ou d'écrire dans un fichier, ...

Supposons que le contenu du fichier *myscript.sce* soit le suivant :

```
disp("Hello World !")
```

Dans la console *Scilab*, nous pouvons utiliser la fonction *exec* pour exécuter le contenu de ce script :

```
-->exec("myscript.sce")
-->disp("Hello World !")
 Hello World !
```

1.7 Commentaires

Comme déjà signalé, toute ligne qui commence par deux barres obliques // est considérée par *Scilab* comme un commentaire et est ignorée.

```
// Mon commentaire
```

Dans l'éditeur de *Scilab*, un commentaire est affiché avec une couleur et une fonte spécifique.
Il n'y a pas de possibilité de commenter un bloc de lignes, comme avec la syntaxe """ ... """ du langage *Python*.
Toutefois, il est possible de commenter un bloc de plusieurs lignes de code en une seule fois, à l'aide de l'éditeur. Cette fonctionnalité est disponible dans le menu "Format" de l'éditeur :

- commenter la sélection (*Ctrl+D*),
- décommenter la sélection (*Ctrl+Maj+D*).

2

Scilab en mode calculatrice

2.1 Types de données

Il faut savoir avant toute chose que tous les objets ou données *Scilab* sont des matrices de différents types : scalaires, booléens, chaînes de caractères, polynômes, matrices, listes, ...
Il existe des données prédéfinies et d'autres qui sont définies par l'utilisateur qui affecte des valeurs à différentes variables.

`type(x)` renvoie un entier donnant le code numérique du type de x défini comme suit :

code	type
1	matrice réelle ou complexe
2	matrice polynomiale
4	matrice booléenne
5	matrice creuse
6	matrice creuse booléene
7	matrice creuse *Matlab*
8	matrice d'entiers codés sur 1, 2 ou 4 octets
9	matrice de "handle" sur les objets graphiques
10	matrice de chaînes de caractères
11	fonction non-compilée (Code *Scilab*)
13	fonction compilée (Code *Scilab*)
14	fonction d'une bibliothèque
15	liste
16	liste typée (tlist)
17	liste typée orientée matrice (mlist)
128	pointeur
129	polynôme implicite utilisé pour indexer
130	fonction *Scilab* intrinsèque (Code *C* ou *Fortran*)

TABLE 2.1 – Types de données

On peut par exemple convertir des chaînes de caractères en scalaires avec la commande `evstr` et vice-versa avec la commande `string` :

```
-->a=12,type(a),b=string(a),type(b),c=evstr(a),type(c),d=evstr(b),type(d)
 a  =

    12.
 ans  =

    1.
 b  =

 12
 ans  =

    10.
 c  =

    12.
 ans  =

    1.
 d  =

    12.
 ans  =

    1.
```

2.2 Les nombres

Les objets de base du langage *Scilab* sont les nombres réels. Il sont représentés par des nombres flottants à double précision (c'est-à-dire avec un exposant signé et une mantisse d'environ 16 décimales). *Scilab* les affiche dans la console par défaut sous forme décimale avec 8 chiffres significatifs.
Notez qu'un nombre en *Scilab* contient toujours le caractère " . ", y compris les nombres entiers (même si le point est facultatif lors de la saisie).

2.2.1 Opérations

Les opérations usuelles sur les réels sont codées de manière naturelle :

addition	soustraction	multiplication	division	puissance
+	−	*	/	^ou **

TABLE 2.2 – Opérations simples

et on a en plus accès à la plupart des fonctions spéciales :

fonction	description
`sin()`	sinus avec argument en radians
`sind()`	sinus avec argument en degrés
`asin()`	arcinus en radians
`asind()`	arcinus en degrés
`cos()`	cosinus avec argument en radians
`cosd()`	cosinus avec argument en degrés
`acos()`	arcosinus en radians
`acosd()`	arcosinus en degrés
`tan()`	tangente avec argument en radians
`tand()`	tangente avec argument en degrés
`atan()`	arctangente en radians
`atand()`	arctangente en degrés
`cotg()`	cotangente avec argument en radians
`cotd()`	cotangente avec argument en degrés
`acot()`	arccotangente avec argument en radians
`acotd()`	arccotangente avec argument en degrés

TABLE 2.3 – Fonctions trigonométriques

fonction	description
`sinh()`	sinus hyperbolique
`asinh()`	arcsinus hyperbolique
`cosh()`	cosinus hyperbolique
`acosh()`	arccosinus hyperbolique
`tanh()`	tangente hyperbolique
`atanh()`	arctangente hyperbolique
`coth()`	cotangente hyperbolique
`acoth()`	arccotangente hyperbolique

TABLE 2.4 – Fonctions hyperboliques

fonction	description
`round()`	arrondi
`int()`	troncature
`floor()`	partie entière
`ceil()`	partie entière supérieure

TABLE 2.5 – Fonctions d'arrondis

fonction	description
`exp()`	exponentielle
`log()`	logarithme népérien
`log10()`	logarithme décimal
`log2()`	logarithme en base 2

TABLE 2.6 – Logarithmes et exponentielle

fonction	description
`sqrt()`	racine carrée
`abs()`	valeur absolue
`modulo()`	reste de la division entière

TABLE 2.7 – Autres fonctions

Pour d'autres fonctions ou pour un descriptif plus complet, on pourra visiter la page :
http://help.scilab.org/docs/5.3.0/fr_FR/section_374347a85fbb0c7f72dfce259ad4ab7a.html.

2.2.2 Exemples

Avec les fonctions précédentes, on obtient par exemple :

```
-->cotg(45)
 ans  =

    0.6173696

-->coth(45)
 ans  =

    1.

-->acoth(2)
 ans  =

    0.5493061

-->round(4.5)
 ans  =

    5.

-->round(4.49)
 ans  =

    4.

-->int(4.4)
 ans  =

    4.

-->int(-5.6)
 ans  =

  - 5.

-->floor(5.7)
 ans  =

    5.

-->ceil(4.56)
 ans  =

    5.
```

```
-->modulo(9,2)
 ans  =

    1.

-->modulo(13,3)
 ans  =

    1.

-->abs(-6.8)
 ans  =

    6.8

-->sqrt(7)
 ans  =

    2.6457513

-->sqrt(-7)
 ans  =

    2.6457513i
```

2.2.3 Constantes

Les constantes, comme e et π, jouent un rôle particulier en mathématiques : elles ont donc une place à part dans *Scilab* et sont stockées dans des *variables non modifiables* qui commencent par le caractère %. Voici les plus utiles :

- %e ,%pi sont assez explicites ($exp(1)$ et π),
- la constante *%i* représente le nombre complexe i : *Scilab* manipule les nombres complexes exactement comme les nombres réels, comme ce sera vu dans le paragraphe sur les complexes page 19,
- %eps représente le "zéro machine" : %eps donne l'ordre de grandeur des erreurs d'arrondis dans les calculs en nombres flottants,
- %inf représente "l'infini machine" : toute valeur trop grande (grosso-modo plus grande que 10^{300}) est considérée comme ayant la valeur %inf (infini),
- quand un résultat ne peut être interprété, *Scilab* renvoie %nan qui signifie "Not A Number",
- Enfin les deux valeurs "Vrai" et "Faux" sont deux constantes définies en *Scilab* par %T (ou %t) et %F (ou %f) : voir plus loin la partie sur les booléens.
- %s est une variable de polynôme : ce sera vu lors de l'étude des matrices et polynômes.

Voici un récapitulatif :

constante	valeur
$\%pi$	3.1415927
$\%e$	2.7182818
$\%i$	$\sqrt{-1}$
$\%eps$	précision machine
$\%inf$	infini
$\%t$	vrai
$\%f$	faux
$\%s$	variable de polynôme

TABLE 2.8 – Constantes prédéfinies

```
-->(1+%i)^2
 ans  =

    2.i

-->%eps
 %eps  =

    2.220D-16

-->1+%eps
 ans  =

    1.

-->10^308 // ou 10**308
 ans  =

    1.00D+308

-->10^309
 ans  =

    Inf

-->%inf*0
 ans  =

    Nan

-->%inf*0
 ans  =
   Nan
```

2.2.4 Les complexes

Comme nous l'avons aperçu précédemment, *Scilab* manipule les nombres complexes : les opérations algébriques usuelles peuvent être effectuées.

```
-->z1=3+4*%i
 z1  =

    3. + 4.i
```

```
-->z1=3+4*%i;

-->z2=2+%i;

-->z1+z2
 ans  =

    4. + 2.i

-->z1*z2
 ans  =

    3. + 4.i
```

D'autres fonctions sont également prédéfinies pour les complexes :

fonction	description
`real`	partie réelle
`imag`	partie imaginaire
`conj`	complexe conjugué
`abs`	module
`phasemag`	argument (en degrés)

TABLE 2.9 – Fonctions complexes

2.3 Les variables

Pour simplifier la manipulation des nombres, nous allons utiliser des variables. Une variable est déterminée par un nom composé d'une suite d'au plus 24 caractères commençant par une lettre éventuellement suivie d'autres lettres, de chiffres, ou de caractères spéciaux comme "_" mais différents des opérateurs déjà définis par *Scilab* : `+*/^[](){}'&|~` .
L'affectation d'une valeur (réelle, complexe, booléenne ou autre) dans une variable se fait en utilisant le signe d'égalité =.
Si le résultat de la dernière opération n'a pas été stocké dans une variable spécifique, alors il l'est dans la variable `ans` :

```
-->a=1+%i;b=sqrt(2); c=a/b
 c  =

 .7071068 + .7071068i
-->d=%f
 d  =

F
-->a*b
 ans  =

 1.4142136 + 1.4142136i
-->ans
 ans  =

 1.4142136 + 1.4142136i
```

Il n'y a pas de déclaration préalable des variables en *Scilab* : celles-ci sont créées lors de l'affectation. La notion de type d'une variable (`constant, string, function ...`) est donc beaucoup moins rigide en *Scilab* que dans un langage compilé. En particulier, le type d'une variable donnée peut changer d'une affectation à l'autre.

Si besoin, on peut tester le type d'une variable avec la fonction `typeof` :

```
 -->a,typeof(a)
 a  =

    1. + i
 ans  =

 constant
-->d,typeof(d)
 d  =

  F
 ans  =

 boolean
```

Plusieurs commandes permettent de connaître la liste des variables utilisées dans un session *Scilab*. La commande `who` liste tous les noms des variables de l'environnement *Scilab*. On peut ainsi se rendre compte que *Scilab* travaille avec de nombreuses variables d'environnements (chemins vers des répertoires particuliers, noms de librairies, constantes particulières, ...).
La commande `whos -type` permet de spécifier le type de variables qu'on veut lister, et donc de clarifier quelque peu l'affichage. Il existe aussi une interface graphique pour visualiser la liste des variables et leur type. On y accède par le menu *Applications->Navigateur de variables*.

```
->who

Vos variables sont :

               %h_delete          get_figure_handle                subdemolist
   resize_demo_gui
              resize_gui            demo_gui_update              create_frame
          demo_gui
                    help                       home       external_objectslib
modules_managerlib
            helptoolslib               scinoteslib                   xcoslib
          matiolib
                atomslib               atomsguilib             parameterslib
simulated_annealinglib
   genetic_algorithmslib                umfpacklib           scicos_autolib
   scicos_utilslib
          spreadsheetlib             demo_toolslib                 assertlib
development_toolslib
                soundlib                 tclscilib                 m2scilib
compatibility_functilib
              arnoldilib             statisticslib                   timelib
         stringlib
    special_functionslib                 sparselib      signal_processinglib
                %z
                     %s            polynomialslib            overloadinglib
   optimsimplexlib
             optimbaselib             neldermeadlib          optimizationlib
 linear_algebralib
```


```
                 jvmlib         output_streamlib                     iolib
  interpolationlib
             integerlib          dynamic_linklib                    guilib
        uitreelib
     data_structureslib                  cacsdlib        graphic_exportlib
      graphicslib
            datatipslib                 fileiolib              functionslib
elementary_functionslib
differential_equationlib                  corelib                      PWD
               %tk
                     %F                        %T                     %nan
              %inf
                    SCI                   SCIHOME                   TMPDIR
              %gui
                  %fftw                         $                       %t
                %f
                   %eps                       %io                       %i
                %e
                    %pi

 utilise      14705 éléments parmi   10000000.
 et         85 variables parmi       9231.

Vos variables globales sont :

           %modalWarning              %toolboxes            %toolboxes_dir
            %helps
                demolist             demolistlock

 utilise        640 éléments parmi      10999.
 et            6 variables parmi       767.

-->whos
Nom                      Type           Dimensions     Octets

%e                       constant       1  par 1       24
%eps                     constant       1  par 1       24
%F                       boolean        1  par 1       24
%f                       boolean        1  par 1       24
%fftw                    boolean        1  par 1       24
%gui                     boolean        1  par 1       24
%h_delete                function                      432
%helps                   constant*      0  par 0       16
%i                       constant       1  par 1       32
%inf                     constant       1  par 1       24
%io                      constant       1  par 2       32
%modalWarning            boolean*       1  par 1       24
%nan                     constant       1  par 1       24
%pi                      constant       1  par 1       24
%s                       polynomial     1  par 1       56
%T                       boolean        1  par 1       24
%t                       boolean        1  par 1       24
%tk                      boolean        1  par 1       16
%toolboxes               string*        1  par 1       40
%toolboxes_dir           string*        1  par 1       136
%z                       polynomial     1  par 1       56
arnoldilib               library                       264
assertlib                library                       576
atomsguilib              library                       344
atomslib                 library                       1000
cacsdlib                 library                       4072
compatibility_functilib  library                       4144
```

corelib	library		592
create_frame	function		12040
data_structureslib	library		464
datatipslib	library		1096
demo_gui	function		11456
demo_gui_update	function		7152
demo_toolslib	library		616
demolist	string*	15 par 2	4880
demolistlock	constant*	0 par 0	16
development_toolslib	library		544
differential_equationlib	library		496
dynamic_linklib	library		744
elementary_functionslib	library		2960
external_objectslib	library		304
fileiolib	library		672
functionslib	library		728
genetic_algorithmslib	library		672
get_figure_handle	function		2152
graphic_exportlib	library		296
graphicslib	library		3920
guilib	library		488
help	function		5368
helptoolslib	library		728
home	string	1 par 1	72
integerlib	library		1416
interpolationlib	library		336
iolib	library		440
jvmlib	library		296
linear_algebralib	library		1400
m2scilib	library		352
matiolib	library		328
modules_managerlib	library		704
neldermeadlib	library		1144
optimbaselib	library		1000
optimizationlib	library		744
optimsimplexlib	library		1200
output_streamlib	library		312
overloadinglib	library		15904
parameterslib	library		424
polynomialslib	library		928
PWD	string	1 par 1	72
resize_demo_gui	function		7200
resize_gui	function		3608
SCI	string	1 par 1	96
scicos_autolib	library		360
scicos_utilslib	library		504
SCIHOME	string	1 par 1	160
scinoteslib	library		296
signal_processinglib	library		2056
simulated_annealinglib	library		600
soundlib	library		544
sparselib	library		504
special_functionslib	library		304
spreadsheetlib	library		280
statisticslib	library		1384
stringlib	library		696
subdemolist	string	12 par 2	4360
tclscilib	library		360
timelib	library		520
TMPDIR	string	1 par 1	120
uitreelib	library		512
umfpacklib	library		456

```
whos                    function               15416
xcoslib                 library                808
```

Subtilité de `who` :

```
--> a=1
--> A=2
--> whos() \\ les détails apparaîssent
--> clear a \\
--> who \\a disparaît
--> clear
--> who \\ a, A et ans disparaîssent
```

2.4 Les booléens

Scilab permet aussi d'effectuer les principales opérations de comparaisons entre nombres. Le résultat d'une telle opération est un booléen. *Scilab* reconnaît donc aussi le type *booléen* et effectue les opérations élémentaires associées. Les deux valeurs "Vrai" et "Faux" sont deux constantes définies en *Scilab* par `%T` (ou `%t`) et `%F` (ou `%f`).
Le tableau ci-dessous donne les principaux opérateurs associés aux booléens :

égal	différent	inférieur strict	supérieur strict	inférieur ou égal	supérieur ou égal	ou	et	non
==	<> ou ~=	<	>	<=	>=	|	&	~

TABLE 2.10 – Booléens

La fonction `bool2s` convertit des booléens en 0 (pour `F`) et en 1 (pour `T`).

⚠ Ne pas confondre l'opérateur de comparaison == avec celui d'affectation =.
Voici quelques exemples d'utilisation des booléens avec *Scilab* :

```
-->3<1
ans =

 F
-->3=1//mauvaise comparaison
 Warning: obsolete use of =
instead of == !

 ans  =
  F

-->c=3>1
 c  =

  T

-->bool2s(c)
 ans  =

    1.

-->3==1//bonne comparaison
 ans  =
  F
```

```
-->%i^2==-1
 ans =

 T
-->~(3<1)&(%i^2==-1)
 ans =

 T
-->%T|%F
 ans =
 T

-->suite1=3:2:15
 suite1  =

    3.    5.    7.    9.    11.    13.    15.

-->suite1>7
 ans  =

  F F F T T T T
```

On peut fabriquer des matrices de booléens à partir de comparaisons directement entre matrices de réels. La fonction `find` permet de trouver les cases de valeur *vrai* dans une matrice de booléens :

```
-->x=rand(2,4)
 x  =
    0.8782165    0.5608486    0.7263507    0.5442573
    0.0683740    0.6623569    0.1985144    0.2320748
-->y=rand(2,4)
 y  =
    0.2312237    0.8833888    0.3076091    0.2146008
    0.2164633    0.6525135    0.9329616    0.312642

-->x<y
 ans  =
  F T F F
  T F T T
-->min(x)//minimum de x
 m  =
    0.0683740

-->x==min(x)
 ans  =
  F F F F
  T F F F

-->[i,j]=find(x==min(x))//position du minimum
 j  =

    1.
 i  =

    2.

-->[i,j]=find(x==1)//un cas sans solution
 j  =

     []
 i  =
```

```
    []
```

2.5 Les chaînes de caractères

2.5.1 Généralités

Scilab permet de manipuler facilement les chaînes de caractères. Pour créer une chaîne de caractères, il suffit de la mettre entre apostrophes :

```
-->'une chaîne'
 ans  =

 une chaîne
```

Si on veut mettre une apostrophe (ou un guillemet) dans une chaîne de caractères, il faudra la (le) faire précéder elle-même (lui-même) d'une apostrophe (ou d'un guillemet) :

```
-->txt='le caractère d''échappement est l''apostrophe'
 txt  =

 le caractère d'échappement est l'apostrophe

-->txt='le caractère d''échappement est l"'apostrophe'
 txt  =

 le caractère d'échappement est l'apostrophe

-->txt='des guillemets ""'
 txt  =

 des guillemets "

-->txt='des guillemets '"'
 txt  =

 des guillemets "
```

On peut effectuer de nombreuses opérations sur les chaînes : concaténer deux chaînes par $+$, calculer la longueur d'une chaîne par `length()` ou encore extraire le $k^{\text{ème}}$ élément d'une chaîne avec `part(chaîne,k)` :

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

    43.
-->part(txt,2)
 ans  =

 e
-->'texte='+txt
 ans  =

 texte=le caractère d'échappement est l'apostrophe
```

Chaque caractère possède un code numérique. Les fonctions `str2code` et `code2str` permettent de passer de la chaîne au code et inversement. Par exemple le code de *a* est 10 et 11 est le code de *b* :

```
-->str2code('a'),code2str(11)
 ans  =

    10.
 ans  =

 b
-->logo='Scilab 5'
 logo  =

 Scilab 5
-->str2code(logo).'
 ans  =

  - 28.    12.    18.    21.    10.    11.    40.    5.
-->code2str(ans)
 ans  =

 Scilab 5
```

La conversion d'un résultat de calcul en chaînes de caractères se fait par la fonction `string`.

⚠ Ne pas confondre la chaîne affichée et la variable qu'elle représente :

```
-->%e//ne pas confondre le nombre
 %e  =

    2.7182818

-->string(%e)//et la chaîne de caractères
 ans  =

 2.7182818

-->%e==string(%e)
 ans  =

  F
```

2.5.2 Affichage

Lors de l'exécution de certains programmes, il peut être utile de faire apparaître des informations à l'écran. Pour cela, on peut utiliser la fonction `disp` :

```
-->u=%e-1
 u  =

    1.7182818
-->disp(u)//affichage d'une variable

    1.7182818
-->disp('u=%e-1='+string(u))//affichage d'un message

 u=%e-1=1.7182818
-->disp(u,'u=%e-1=')//plus complexe

 u=%e-1=
    1.7182818
```

On peut faire appel à d'autres foncions pour afficher des résultats à l'écran. La fonction `printf` peut par exemple être appelée directement dans *Scilab* :

```
-->printf('la valeur de e est \n e= %f \n',%e)
la valeur de e est
 e= 2.718282
-->printf('les premiers entiers naturels sont %d,%d,%d ...\n',0,1,2)
les premiers entiers naturels sont 0,1,2 ...
```

D'autres exemples dans le script suivant :

input1.sce

```
1  printf('un entier : %d\n', 10);
2  printf('un autre entier : %i\n', 10);
3  printf('un entier décalé à droite : %4d\n', 10);
4  printf('un entier justifié à gauche : %-4d\n', 10);
5  printf('un entier converti en réel : %#f\n',10);
6  printf('un entier avec un signe + : %+4d\n', 10);
7  printf('un autre entier avec un signe - : %+4d\n', -10);
8  printf('un entier  avec des zéros devant : %06d\n', 10);
9  printf('un entier converti en hexadécimal : %x\n', 10);
10 printf('un réel converti en entier : %d\n', %pi);
11 printf('un réel arrondi : %3.3f\n', %pi);
12 printf('un autre réel arrondi : %2.5g\n', %pi);
13 printf('un réel converti en entier avec des zéros devant : %2.5d\n',
   %pi);
14 printf('un réel (sous forme exponentielle) : %2.5e\n', %pi);
15 printf('un caractère : %c\n', 'a');
16 printf('un caractère : %c\n', 'aaa');
17 printf('une chaîne : %s\n', 'Scilab');
18
19 a = input("Entrez un entier a : ");
20 b = input("Entrez un réel b : ");
21 ch = input("Entrez une chaîne ch : ");
22
23 printf("%d est  un entier, %f est un réel, %s est une chaîne", a, b,
   ch)
```

qui donne :

```
-->exec('/media/marco3/Données 1,9To/Informatique/Info_marco/eleve-scilab/input1.
sce', -1)
un entier : 10
un autre entier : 10
un entier décalé à droite :   10
un entier justifié à gauche : 10
un entier converti en réel : 10.000000
un entier avec un signe + :  +10
un autre entier avec un signe - :  -10
un entier  avec des zéros devant : 000010
un entier converti en hexadécimal : a
un réel converti en entier : 3
un réel arrondi : 3.142
un autre réel arrondi : 3.1416
un réel converti en entier avec des zéros devant : 00003
un réel (sous forme exponentielle) : 3.14159e+00
un caractère : a
un caractère : a
une chaîne : Scilab
Entrez un entier a : 25
```

```
Entrez un réel b : 23
Entrez une chaîne ch : uygjh
Variable non définie : uygjh
Entrez une chaîne ch : 'hgjg'
25 est  un entier, 23.000000 est un réel, hgjg est une chaîne
```

Sans entrer dans les détails, on peut citer les possibilités suivantes (aller voir l'aide pour plus de précision) :

- %d et %i pour les entiers,
- %f pour les réels en précisant le nombre de chiffres avant et après la virgule,
- %e pour les réels sous forme exponentielle (puissances de 10) en précisant le nombre de chiffres après la virgule,
- %g pour les réels en précisant le nombre total de chiffres significatifs.

2.6 Les matrices

2.6.1 Généralités

Comme déjà dit précédemment, il s'agit de la structure de données la plus importante dans *Scilab* : elle sera constamment utilisée. *Scilab* permet de construire toutes sortes de matrices, à partir des réels, mais aussi en utilisant des booléens, des chaînes de caractères ... On peut ensuite effectuer toutes les opérations algébriques usuelles définies sur les matrices.

Il y a plusieurs manières de définir une matrice. La plus simple consiste à saisir la liste des cœfficients d'une matrice ligne par ligne, les lignes étant séparées par des points-virgules, en utilisant l'opérateur de concaténation `[ ]` :

```
-->A=[1 2 3; 4 5 6; 7 8 9]
 A =
 1. 2. 3.
 4. 5. 6.
 7. 8. 9.

--> B=[-1 0; 0 1; -3 0]
 B =
 - 1. 0.
   0. 1.
 - 3. 0.
```

Remarque

On peut mettre une virgule à la place d'un espace pour séparer les éléments d'une même ligne.

On peut accéder aux valeurs stockées dans une matrice en utilisant l'opérateur d'extraction `()`. Pour récupérer la valeur en position i,j de la matrice `A` il suffit d'appeler la commande `A(i,j)` sauf pour la matrice vide `[]` et éventuellement les matrices à une ligne ou une colonne. Pour modifier la valeur en position i, j de la matrice `A`, on peut effectuer par exemple `A(i,j)=...`.

```
-->A(1,2)
 ans  =

    2.
-->A(1,2)=-2
 A  =

    1.  - 2.    3.
    4.    5.    6.
    7.    8.    9.
```

```
-->L=[1 2 3]//la matrice ligne
 L  =

    1.    2.    3.
-->L(1,3)//accès normal
 ans  =
    3.
-->L(2)//accès simplifié
 ans  =
    2.
```

Lorsqu'on modifie un cœfficient qui "déborde" de la matrice de départ, la matrice est augmentée du nombre de lignes et de colonnes nécessaires, et les cases correspondantes sont mises à 0 :

```
-->C=[1 2 3; 4 5 6; 7 8 9]
 C  =

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



-->C(5,4)=10
 C  =

    1.    2.    3.    0.
    4.    5.    6.    0.
    7.    8.    9.    0.
    0.    0.    0.    0.
    0.    0.    0.    10.
```

Pour de petites matrices, on rentrera tous les cœfficients un à un comme ci-dessus mais pour des matrices plus importantes, il est utile de pouvoir les construire plus ou moins automatiquement. On peut utiliser pour cela l'opérateur d'incrémentation symbolisé par " : ". Par exemple, pour construire une matrice à une ligne contenant les entiers de `a` à `b`, on écrira `[a : b]`. Si on veut spécifier un *pas* différent de 1, on pourra écrire `[a : pas : b]` avec *pas* adapté (2 pour aller de 2 en 2 ou -0.5 pour diminuer de $0,5$ à chaque pas) :

```
-->L1=[1:5]
 L1  =

    1.    2.    3.    4.    5.

-->L2=[9:2:16]
 L2  =

    9.    11.    13.    15.

-->L3=[10:-0.5:7]
 L3  =

    10.    9.5    9.    8.5    8.    7.5    7.
-->L=[]
 L  =

      []
```

On peut ensuite construire d'autres matrices à partir des précédentes par concaténation :

```
-->[L1, L2]//on place L1 à gauche de L2
 ans  =
    1.    2.    3.    4.    5.    7.    9.    11.    13.    15.
```

```
-->[L1; L2]//on place L1 au-dessus de L2
 ans  =
    1.    2.    3.    4.    5.
    7.    9.    11.   13.   15.
```

mais il faut faire attention aux tailles des matrices utilisées :

```
-->[A, B] //juxtaposer A puis B
 ans  =
    1.   - 2.    3.  - 1.    0.
    4.     5.    6.    0.    1.
    7.     8.    9.  - 3.    0.

-->[A;B]//pour mettre B sous A
     !--error 6
inconsistent row/column dimensions

-->[B,A]//juxtaposer B puis A

 ans  =

  - 1.    0.    1.   - 2.    3.
    0.    1.    4.     5.    6.
  - 3.    0.    7.     8.    9.
```

2.6.2 Matrices pré-remplies

Il existe aussi des fonctions permettant de créer des matrices de tailles (p,n) déjà remplies :

commande	type de matrice
`[]`	matrice vide
`zeros(n,p)`	matrice nulle
`eye(n,p)`	matrice identité de taille (n,p) complétée par des 0
`ones(n,p)`	matrice de 1
`rand(n,p)`	matrice aléatoire
`diag(v)`	matrice diagonale définie par le vecteur v
`toeplitz`	matrice à diagonale constante
`testmatrix`	carrés magiques et autres

TABLE 2.11 – Matrices préremplies

On peut aussi appliquer ces différentes fonctions à des matrices déjà existantes pour obtenir une matrice de même taille (essayez `zeros(A)`, `zeros(B)`).

```
-->zeros(4,3)
 ans  =

    0.    0.    0.
    0.    0.    0.
    0.    0.    0.
    0.    0.    0.
-->ones(2,3)
 ans  =

    1.    1.    1.
    1.    1.    1.
```

```
-->eye(3,3)
 ans  =

    1.    0.    0.
    0.    1.    0.
    0.    0.    1.
-->rand(3,2)
 ans  =

    0.9488184    0.7340941
    0.3435337    0.2615761
    0.3760119    0.4993494
```

2.6.3 Opérations sur les matrices

La syntaxe des opérations matricielles est plus compliquée que pour les nombres flottants. En effet, il faut distinguer :

- les opérations "terme à terme" qui s'appliquent à des matrices de même taille,
- les opérations matricielles pour lesquelles les tailles des matrices en présence doivent vérifier certaines conditions,
- les opérations qui combinent une matrice et un nombre.

2.6.3.1 Opérations terme à terme

Commençons par les opérations terme à terme (par ordre de priorité croissante) :

addition	soustraction	multiplication	division	puissance
+	-	.*	./	.^

TABLE 2.12 – Opérations terme à terme sur les matrices

```
-->A+A
 ans  =

    2.   - 4.     6.
    8.     10.    12.
    14.    16.    18.
-->A-A
 ans  =

    0.    0.    0.
    0.    0.    0.
    0.    0.    0.
-->A./A
 ans  =

    1.    1.    1.
    1.    1.    1.
    1.    1.    1.
-->A.*A
 ans  =

 1.     4.     9.
 16.    25.    36.
 49.    64.    81.
```

```
-->A.^2
 ans  =

 1.     4.     9.
 16.    25.    36.
 49.    64.    81.
-->A.^A
 ans  =

 1.          0.25          27.
 256.        3125.         46656.
 823543.     16777216.     3.874D+08
```

Ne pas les confondre avec les opérations purement matricielles ci-dessous.

2.6.3.2 Opérations matricielles

Par ordre de priorité croissante, on a :

produit	division	puissance	transposition	conjugaison
*	/	^	.'	'

TABLE 2.13 – Opérations matricielles sur les matrices

```
-->A*B//produit matriciel
 ans  =

  - 10.   - 2.
  - 22.     5.
  - 34.     8.
-->B.'//transposée
 ans  =

  - 1.     0.   - 3.
    0.     1.     0.

-->A*A
 ans  =

    14.      12.     18.
    66.      65.     96.
    102.     98.     150.
-->A^2
 ans  =

    14.      12.     18.
    66.      65.     96.
    102.     98.     150.
```

Il faut donc faire attention à ce que les tailles des matrices soient compatibles entre elles, sinon on obtiendra un message d'erreur :

```
-->A+B
    !--error 8
inconsistent addition
```

```
-->B*A
    !--error 10
inconsistent multiplication
```

2.6.3.3 Opérations avec des matrices 1 x 1

Dans le cas où l'une des matrices est de taille 1×1, on fera une opération terme à terme entre un nombre et une matrice :

```
-->A
 A  =

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

-->1+A
 ans  =

    2.   - 1.    4.
    5.     6.    7.
    8.     9.    10.

-->2-A
 ans  =

    1.     4.   - 1.
  - 2.   - 3.   - 4.
  - 5.   - 6.   - 7.

-->3*A
 ans  =

    3.    - 6.     9.
    12.     15.    18.
    21.     24.    27.
```

2.6.3.4 Fonctions matricielles

Il faudra faire attention aux confusions possibles entre les différentes opérations matricielles, termes à termes ou scalaires. Le cas des divisions matricielles peut notamment conduire à des erreurs complexes : Lorsqu'on veut faire une division terme à terme d'un réel par une matrice il faut faire attention à la position du point du `./`. En effet, s'il touche le réel à diviser, le point sera ignoré : `1./A` $\Leftrightarrow$1/A $\not\Leftrightarrow$`(1)./A` .
Pour une matrice A carrée, `1/A` est en fait l'inverse de la matrice (s'il existe) qu'on peut aussi obtenir par `inv(A)` ou `A^(-1)`.

On peut vérifier la différence sur l'exemple suivant :

```
-->1./A
 ans  =

   0.125   - 1.75          1.125
 - 0.25      0.5         - 0.25
   0.125     0.9166667   - 0.5416667

-->(1)./A
 ans  =
```

```
   1.          - 0.5       0.3333333
   0.25          0.2       0.1666667
   0.1428571     0.125     0.1111111
```

Pour le calcul d'inverse (qui peut ne pas exister dans certains cas), on obtient le même résultat qu'avec `1/A`.

```
-->A^(-1)
 ans  =

   0.125   - 1.75          1.125
 - 0.25      0.5         - 0.25
   0.125     0.9166667   - 0.5416667

-->inv(A)
 ans  =

   0.125   - 1.75          1.125
 - 0.25      0.5         - 0.25
   0.125     0.9166667   - 0.5416667
```

Enfin, quand on a une matrice `A` de taille inconnue, on peut récupérer le nombre de lignes et/ou de colonnes par les instructions `size(A)`,`size(A,1)`,`size(A,2)`. On peut aussi utiliser `length()` quand la matrice possède une seule ligne ou une seule colonne pour récupérer sa longueur.

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

    3.    3.

-->size(B)
 ans  =

    3.    2.



-->size(C)
 ans  =

    5.    4.
-->size(C,1)
 ans  =

    5.
-->size(C,2)
 ans  =

    4.
-->size(L1)
 ans  =

    1.    5.


-->length(L1)
 ans  =

    5.
```

En général, lorsqu'on applique une fonction de $\mathbb{R}$ dans $\mathbb{R}$ à une matrice, elle est appliquée à chaque terme de la matrice. Cependant, certaines fonctions courantes possèdent une définition matricielle que ne correspond pas simplement à l'application de la fonction à chaque terme de la matrice. Dans *Scilab*, les versions

matricielles des fonctions réelles sont désignées par le même nom avec un `m` en plus à la fin. Les autres fonctions sont nombreuses. En voici quelques unes :

fonction	description
`acoshm`	calcule l'arccosinus hyperbolique matriciel inverse
`acosm`	calcule l'arccosinus matriciel inverse
`asinhm`	calcule l'arcsinus hyperbolique matriciel inverse
`asinm`	calcule l'arcsinus matriciel inverse
`atanhm`	calcule l'arctangente hyperbolique matricielle inverse
`atanm`	calcule l'arctangente matricielle inverse
`bdiag(H)`	calcule la matrice diagonale de la matrice H
`cond(H)`	calcule le conditionnement de la matrice H
`coshm`	calcule le cosinus hyperbolique matriciel
`cosm`	calcule le cosinus matriciel
`cothm`	calcule la cotangente hyperbolique matricielle
`det(H)`	calcule le déterminant de la matrice H
`diag(H)`	extrait la diagonale de la matrice H
`expm(H)`	calcule l'exponentielle matricielle de la matrice H
`inv(H)`	calcule l'inverse de la matrice H
`kernel(H)`	calcule le noyau de la matrice H
`linsolve(G,g)`	calcule la solution de $G * x = -g$
`G/g`	calcule la solution de $x * g = G$
`G\g`	calcule la solution de $G * x = g$
`logm(H)`	calcule le logarithme matriciel de H
`min(H)`	retourne le cœfficient le plus petit de la matrice H
`max(H)`	retourne le cœfficient le plus grand de la matrice H
`norm(H)`	calcule la norme de la matrice H 2 par défaut, 'fro' pour frobenius, 'inf' pour ∞
`poly(H)`	calcule le polynôme cractéristique de la matrice H
`prod(H)`	calcule le produit des cœfficients de la matrice H
`spec(H)`	calcule les valeurs propres de la matrice H
`rank(H)`	calcule le rang de la matrice H
`sinhm`	calcule le sinus hyperbolique matriciel
`sinm`	calcule le sinus matriciel inverse
`size(H)`	calcule la taille de la matrice H
`size(H,"r")`	calcule le nombre de lignes de la matrice H
`size(H,"c")`	calcule le nombre de colonnes de la matrice H
`size(H,"*")`	calcule le nombre total d'éléments de la matrice H
`sqrtm(H)`	calcule la racine carré matricielle de la matrice H
`sum(H)`	calcule la somme des cœfficients de la matrice H
svd(H)	décompose la matrice H en valeurs singulières
`tanhm`	calcule la tangente hyperbolique matricielle
`tanm`	calcule la tangente matricielle inverse
`trace(H)`	calcule la trace de la matrice H
`tril(H)`	rend nuls les cœfficients au-dessus de la diagonale de la matrice H
`triu(H)`	rend nuls les cœfficients au-dessous de la diagonale de la matrice H
`H.'` ou *H'*	calcule la transposée de la matrice H

TABLE 2.14 – Autres fonctions sur les matrices

Des exemples d'utilisation :

```
-->H=[1 2 3 ; 4 5 6 ; 7 8 9]
 H  =

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

-->sum(H)
 ans  =

    45.

-->prod(H)
 ans  =

    362880.

-->min(H)
 ans  =

    1.

-->max(H)
 ans  =

    9.

-->det(H)
 ans  =

    6.661D-16

-->cond(H)
 ans  =

    3.813D+16

-->spec(H)
 ans  =

    16.116844
  - 1.116844
  - 1.304D-15

-->inv(H)
Attention :
La matrice est presque singulière ou mal conditionnée. rcond =    1.5420D-18

 ans  =

 10^15 *

  - 4.5035996    9.0071993  - 4.5035996
    9.0071993  - 18.014399    9.0071993
  - 4.5035996    9.0071993  - 4.5035996

-->trace(H)
 ans  =

    15.
```

```
-->rank(H)
 ans  =

    2.

-->size(H)
 ans  =

    3.    3.

-->poly(H,'s')
 ans  =

                           2   3
  - 2.347D-14 - 18s - 15s + s

-->norm(H)
 ans  =

    16.848103

-->norm(H,'fro')
 ans  =

    16.881943

-->norm(H,'inf')
 ans  =

    24.

->J=[1 2 3 ; 4 5 6]
 J  =

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

-->size(J)
 ans  =

    2.    3.

-->size(J,"r")
 ans  =

    2.

-->size(J,"c")
 ans  =

    3.

-->size(J,"*")
 ans  =

    6.

-->svd(G)
 ans  =

    9.508032
    0.7728696
```

```
-->H=[1 2 3 ; 4 5 6;7 8 9]
 H  =

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

-->bdiag(H)
 ans  =

    16.116844    0.          0.
    0.         - 1.116844    0.
    0.           0.        - 8.046D-16

-->kernel(H)
 ans  =

    0.4082483
  - 0.8164966
    0.4082483

-->H.'
 ans  =

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

-->diag(H)
 ans  =

    1.
    5.
    9.

-->G=[1 2 3 ; 4 5 6]
 G  =

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

-->g=[3 5 9]
 g  =

    3.    5.    9.

-->G/g
 ans  =

    0.3478261
    0.7913043

-->h=[4 ; 7]
 h  =

    4.
    7.

-->(G\h)
 ans  =
```

```
  - 0.5
    0.
    1.5

-->linsolve(G,h)
 ans  =

    0.6666667
  - 0.3333333
  - 1.3333333
```

2.6.3.5 Extraction de matrices et de vecteurs

commande	description	exemple si besoin
`H(i,j)`	cœfficient d'ordre i,j de H	`H(3,4)` donne le coefficient $H_{3,4}$
`H(i1:i2,:)`	lignes $i1$ à $i2$ de H	`H(3:5,:)` donne les lignes 3 à 5
`H($,:)`	dernière ligne de H	
`H(i1:i2,:) = [ ]`	supprime les lignes $i1$ à $i2$ de H	`H(3:5,:) = [ ]` supprime les lignes 3 à 5
`H(:,j1:j2)`	colonnes $j1$ à $j2$ de H	`H(:,4:6)` donne les colonnes 4 à 6
`H(:,$)`	dernière colonne de H	
`H(:,j1:j2) = [ ]`	supprime les colonnes $j1$ à $j2$ de H	`H(:,4:6) = [ ]` supprime les colonnes 4 à 6
`H(i1:i2,j1:j2)`	donne la matrice extraite de $i1$ à $i2$ et de $j1$ à $j2$ de H	`H(3:5,4:6)` donne $H_{i,j}$, i variant de 3 à 5 et j variant de 4 à 6

TABLE 2.15 – Extraction de matrices

commande	description	exemple si besoin
`x:y`	nombres de x à y par pas de 1	`3:5` donne 3, 4, 5
`x:p:y`	nombres de x à y par pas de p	`3:2:9` donne 3, 5, 7, 9
`linspace(x,y,n)`	n nombres entre x et y	`linspace(15,2,5)` donne 15, 11.75, 8.5, 5.25, 2
`v(i)`	$i^{\text{ème}}$ coordonnée de v	`v(3)` donne la coordonnée v_3
`v($)`	dernière coordonnée de v	
`v(i1:i2)`	coordonnées $i1$ à $i2$ de v	`v(3:5)` donne les coordonnées de v_3 à v_5
`v(i1:i2) = [ ]`	supprime les coordonnées de $i1$ à $i2$ de v	`v(3:6) = [ ]` supprime les coordonnées de v_3 à v_6

TABLE 2.16 – Extraction de vecteurs

2.7 Les polynômes et les fonctions réelles

2.7.1 Polynômes simples

L'objet polynôme existe en *Scilab*. On peut le définir directement de deux manières :

- Par défaut, un polynôme est construit à partir de ses racines (ici 1 et 2) :

```
-->p=poly([1 2],'s')
 p  =

              2
   2 - 3s + s
```

- On peut aussi le construire à partir de ses cœfficients (méthode classique) :

```
-->q=poly([1 2],'s','c')
 q  =

   1 + 2s
```

2.7.2 Opérations simples sur les polynômes

Les opérations usuelles avec les polynômes sont prévues par *Scilab* :

```
-->p+q
 ans  =

            2
   3 - s + s

-->p-q
 ans  =

             2
   1 - 5s + s

-->p*q
 ans  =

              2    3
   2 + s - 5s + 2s

-->p/q
 ans  =

              2
   2 - 3s + s
   ----------
     1 + 2s
```

On peut définir en *Scilab* une variable s comme étant le polynôme égal à s :

```
-->z=poly(0,'s')
 z  =

   s
```

2.7.3 Opérations complexes sur les polynômes

La recherche des racines d'un polynôme est importante. Dans *Scilab* est intégrée une procédure numérique qui permet de déterminer, de manière approchée, les racines sous forme de vecteur colonne :
Il s'agit de la commande `roots`. Exemple :

```
-->p1=poly([2 3 4],'x','c')
 p1  =

                2
    2 + 3x + 4x

-->roots(p1)
 ans  =

  - 0.375 + 0.5994789i
  - 0.375 - 0.5994789i
```

Comme on peut le voir, cela fonctionne aussi lorsqu'il existe des racines complexes.

En résumé, quelques opérations importantes sur les polynômes, mais il y en a encore d'autres :

commande	description	exemple
`v=[]`	vecteur	$v = [123]$
`p1 = poly(v,"x")`	polynôme dont les racines sont les éléments de v	`poly(v,"x")` donne $-6+11x-6x^2+x^3$
`p2 = poly(v,"x","c")`	polynôme dont les cœfficients sont les éléments de v	`poly(v,"x","c")` donne $1+2x+3x^2$
`p3 = inv_coeff(v)`	polynôme dont les cœfficients sont les éléments de p	`inv_poly(v)` donne $1+2x+3x^2$
`coeff(p)`	cœfficients de p	`coeff(p1)` donne $-6\ 11\ \ -6\ 1$
`roots(p)`	racines de p	`roots(p1)` donne 3 2 1
`factors(p)`	facteurs irréductibles réels d'un polynôme	`factors(p1)` donne $-3+x$, $-2+x$, $-1+x$
`pol2str(p)`	convertit un polynôme en chaîne de caractères	`pol2str(p1)` donne $-6+11*x-6*x^2+x^3$
`degree(p)`	degré d'un polynôme	`degree(p1)` donne 3

TABLE 2.17 – Manipulation de polynômes

Utilisation de quelques fonctions sur les polynômes :

```
-->p2=poly([1 2 3 4],'x','c')
 p2  =

                2    3
    1 + 2x + 3x + 4x

-->H=[p2 derivat(p2) ; p2-1 derivat(p2)*p1]
 H  =

                2    3                2
    1 + 2x + 3x + 4x     2 + 6x + 12x

            2    3                   2     3     4
    2x + 3x + 4x         4 + 18x + 50x + 60x + 48x

-->derivat(H)
 ans  =

                 2
    2 + 6x + 12x      6 + 24x
```

```
                2                        2      3
  2 + 6x + 12x      18 + 100x + 180x + 192x
```

2.8 Les fonctions

Un des concepts mathématiques les plus importants en informatique est le concept de fonction. *Scilab* possède un type de variable `function` spécifique pour coder les fonctions. En réalité, toutes les commandes que nous avons vues jusqu'ici sont en fait des fonctions *scilab*.

D'un point de vue mathématique, une fonction est une relation qui possède un ou plusieurs paramètres d'entrée et qui associe au plus une valeur en sortie :

$$\begin{array}{rccc} f: & A & \longrightarrow & B \\ & x & \longmapsto & y = f(x) \end{array}$$

La définition d'une fonction dans *Scilab* reprend exactement ce schéma mais sans contrôler le type des variables en entrée/sortie (normal, puisque l'on ne déclare pas les types des variables avant affectation).
La syntaxe utilisée est la suivante :

```
function y=maCommande(paramètres)
  instructions
  y=...
endfunction
```

Par exemple, la fonction de $\mathbb{R}$ dans $\mathbb{R}$ définie par $f(x) = 1 + x^2$ sera codée en *Scilab* par :

```
function [y]=f(x)
y=1+x^2
endfunction
```

Autre exemple :

ttc.sce

```
1  function y=prixTTC(x)
2      y=1.195*x
3  endfunction
4  //Application :
5  monPrix=230.5; // Entrée
6  sortie=prixTTC(monPrix); //Sortie
7  disp(sortie);
```

Maintenant, il faut sauver ce code dans un fichier (appelons le *mafonction.sce*) et le charger dans la session *Scilab* pour pouvoir ensuite l'utiliser. On peut pour cela utiliser n'importe quel éditeur de texte, mais *Scilab* possède un éditeur de texte, *SciNotes*, parfaitement adapté au langage *Scilab* (coloration syntaxique, numérotation des lignes, recherche de mots, interfaçage avec *Scilab*, débogage ...).
Pour lancer *SciNotes*, on peut taper dans la console *editor()* ou bien :

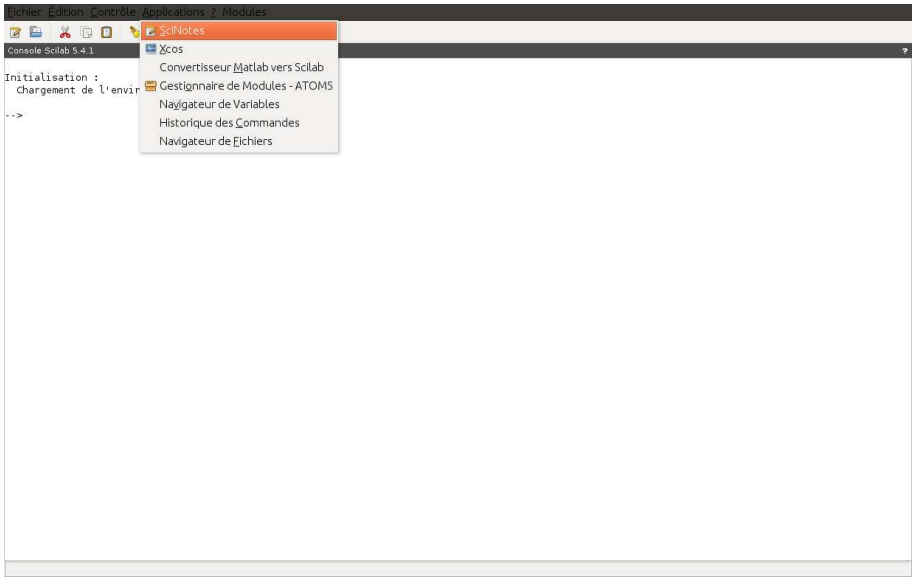


FIGURE 2.1 – Menu applications

- cliquer sur l'onglet *Applications* de la barre de menus comme sur la figure 2.1 ci-dessus,
- dans la console utiliser la commande *SciNotes*.

Une nouvelle fenêtre va s'ouvrir : c'est l'éditeur de texte que vous pouvez utiliser pour saisir le code de la fonction `f` cf. figure 2.2.

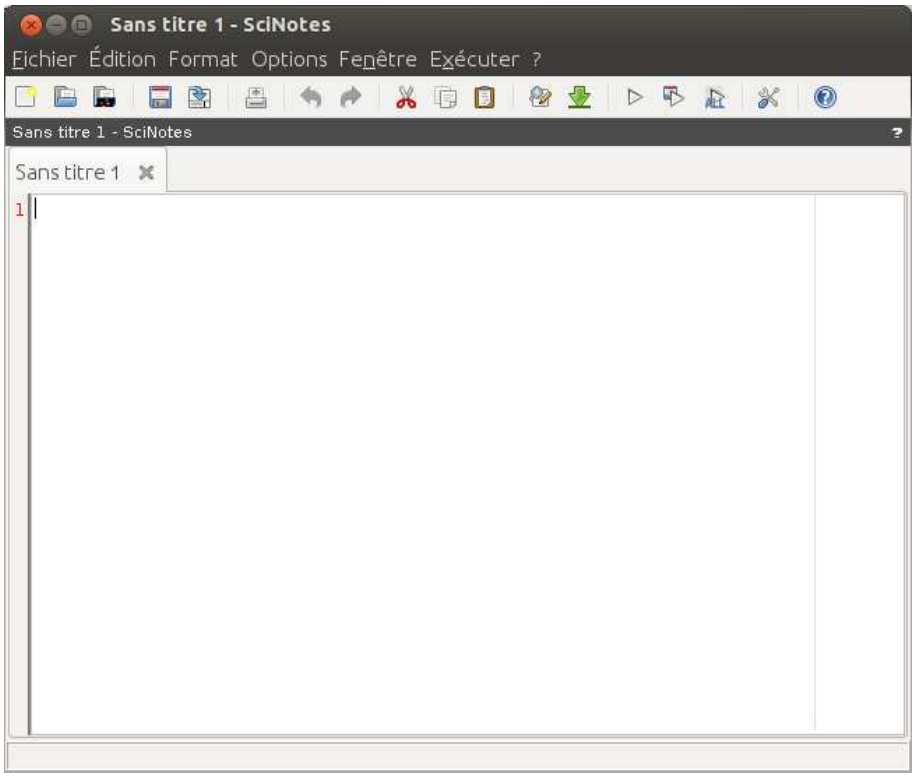


FIGURE 2.2 – Éditeur Scinotes

⚠ Lorsqu'on veut enregistrer des définitions de fonctions *Scilab* dans un fichier, on lui donnera un nom avec une extension de la forme **.sci*. Par contre si on veut sauvegarder dans un fichier une liste de commandes à exécuter séquentiellement (c'est à dire un script), on utilisera plutôt un nom avec une extension de la forme **.sce*.

On peut écrire plusieurs définitions de fonctions dans un même fichier. Une fois le fichier sauvé, on peut le charger dans *Scilab* depuis le menu *Éxécuter* de *SciNotes* ou avec la commande `exec` depuis la console. Une fois chargée, on peut utiliser la fonction comme les autres fonctions *scilab*. On peut aussi avoir besoin de définir une fonction en une seule ligne avec la commande : c'est possible avec la commande `deff`, mais moins maniable qu'avec la commande `function` (il faut une certaine maîtrise des problèmes liés aux chaînes de caractères dans ce cas).

```
-->exec('mafonction.sci')//définition via un fichier
-->deff('[y]=f(x)','y=1+x^2')//définition en ligne
-->f(2)
 ans  =

    5.
-->whos -type function
Name                        Type           Size           Bytes

whos                        function                      8512
f                           function                      216
-->f(x)
f(x)
    !--error 4
undefined variable : x
at line       5 of exec file called by :
exec('fonction.sce',1)
```

L'erreur générée à la dernière ligne ci-dessus amène une remarque importante :

 *Scilab* est un logiciel de calcul numérique et pas un logiciel de calcul formel. En conséquence, la commande `f(x)` n'a aucun sens si x n'est pas une variable définie (x=*2* ou une matrice par exemple). De même, les commandes du type `f(x)=1+x^2` ne définissent pas une fonction *Scilab*.

Si on veut appliquer cette fonction `f` à une matrice il faut se méfier des confusions entre opérations terme à terme et opérations matricielles. Par exemple, si x est une matrice carrée, `x^2` sera interprété comme un produit matriciel et non comme une élévation au carré de chaque terme. Pour éviter ce genre de problèmes on utilisera la commande `feval` pour évaluer une fonction sur chaque valeur d'une matrice :

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

    1.    2.
    3.    4.

-->f(A)
 ans  =

    8.    11.
    16.   23.

-->1+A.^2
 ans  =

    2.    5.
    10.   17.

-->feval(A,f)
 ans  =

    2.    5.
    10.   17.
```

Cependant, en mathématiques, on peut considérer que les ensembles de départ et d'arrivée d'une fonction sont en fait des produits d'ensembles. Ce point de vue est valable dans *Scilab*, ce qui permet de considérer facilement des fonctions ayant plusieurs variables en entrée et en sortie.

⚠ Lorsqu'une fonction renvoie plusieurs valeurs en sortie, il faut impérativement récupérer chacune de ces valeurs dans des variables en suivant la syntaxe d'appel donné dans la première ligne de la fonction :

```
function [y1,y2,y3]=nomfonction(x1,x2,x3)
```

sinon seul le résultat $y1$ sera affiché, les autres résultats du calcul étant perdus.

```
function [a,b]=f(x,y)
a=x+y
b=x*y
endfunction


-->[a,b]=f(2,3)
 b  =
    6.
 a  =
    5.

-->f(2,3)
 ans  =

    5.
```

⚠ Dans la dernière commande, b est perdu car non affecté.

⇒ **Activité 2. 1**

Écrivez une fonction (*renversement.sce*) qui inverse l'ordre des éléments d'une chaîne de caractères.

Le programme :

2.9 Variables locales et globales

Pour bien utiliser les fonctions *Scilab*, il faut aussi comprendre le problème des variables locales et des variables globales. Le mécanisme d'appel de fonction est conçu pour que les variables définies à l'intérieur du corps de la fonction n'interfèrent pas avec celle définies dans la session *Scilab* qui appelle la fonction.

```
-->a=1,b=2
 a  =

    1.
 b  =
```

```
    2.
-->deff('[a,b]=g(x)','a=x+1,b=2*b')
Warning :redefining function: g

-->[u,v]=g(3)//les valeurs de a et b sont prises en compte
 v  =

    4.
 u  =

    4.
-->a,b// elles ne sont pas modifiées dans l'environnement global
 a  =

    1.
 b  =

    2.
```

On retiendra la règle suivante :

Lors de l'appel à une fonction, si une variable apparaissant dans le corps de la fonction n'est pas définie dans l'environnement local (entre les commandes `function` et `endfunction`), alors pour évaluer cette variable, *Scilab* va chercher si cette variable est définie dans la session d'où la fonction a été appelée (ce qu'on appelle l'environnement global). Si la variable n'est pas définie dans l'environnement global, il y aura une erreur avec le message `undefined variable`.

Autant que possible, on évitera d'utiliser dans le corps d'une fonction des variables qui ne sont pas définies dans le corps de la fonction. Si toutefois on a besoin d'utiliser des variables globales, on pourra utiliser la commande `global` pour partager des variables entre plusieurs fonctions et l'environnement global.
La commande `global` permet donc de partager certaines variables en lecture ou en écriture entre des fonctions. Toute affectation à ces variables est propagée à toutes les autres fonctions ayant déclaré cette variable globale (avec ce mot-clé global).
Si la variable n'existe pas au moment où elle est déclarée globale avec le mot-clé `global`, elle est initialisée comme une matrice vide.

3

Programmation

3.1 Boucles et tests

3.1.1 Généralités

Toutes les commandes vues depuis le début de ce chapitre forment les instructions de base du langage de programmation de *Scilab*. Mais pour aller plus loin, il nous manque les instructions de base de l'algorithme que sont les branchements conditionnels et les boucles.

algorithme	scilab
DEBUT_ALGORITHME SI (condition 1) ALORS DEBUT_SI action 1 FIN_SI SINON DEBUT_SINON SI (condition 2) ALORS DEBUT_SI action 2 FIN_SI SINON DEBUT_SINON action 3 FIN_SINON FIN_SINON FIN_ALGORITHME	if condition 1 then action1 elseif condition 2 then action 2 else action 3 end
DEBUT_ALGORITHME POUR *cpt* ALLANT_DE *debut* A *fin* DEBUT_POUR actions FIN_POUR FIN_ALGORITHME	for cpt=debut:pas:fin do \\ si le pas est omis, \\ il vaut 1 actions end
DEBUT_ALGORITHME TANT_QUE (condition) FAIRE DEBUT_TANT_QUE actions FIN_TANT_QUE FIN_ALGORITHME	while condition do actions end

algorithme	scilab
DEBUT_ALGORITHME SUIVANT expression0 DEBUT_SUIVANT CAS cas 1 DEBUT_CAS action 1 FIN_CAS CAS cas 2 DEBUT_CAS action 2 FIN_CAS CAS cas 3 DEBUT_CAS action 3 FIN_CAS SINON DEBUT_SINON action 4 FIN_SINON FIN_SUIVANT FIN_ALGORITHME	select expression0 case cas 1 then action 1 case cas 2 then action 2 case cas 3 then action 3 else action4 end

TABLE 3.2 – Boucles et tests

Dans ce qui précède, *condition* représente une expression dont l'évaluation conduit à une valeur booléenne (%t ou %f) et les *actions* représentent des suites de commandes *Scilab* à exécuter suivant le cas.

On évitera de lancer de lancer des instructions `if`, `for`, `while` ... directement dans la console et on privilégiera l'utilisation de fichiers (scripts). En particulier lorsqu'une commande `if`, `for` ou `while` est lancée dans la console, aucune commande n'est exécutée tant que le `end` correspondant n'a pas été exécuté.

3.1.2 Test conditionnel

Un exemple :

```
-->x=7;if x >= 0 then disp("positif"), else disp("négatif"), end

 positif
```

Dans une structure conditionnelle l'instruction `then` peut être omise, mais si c'est le cas, alors les instructions `if` et `then` doivent obligatoirement être sur la même ligne.
Illustration :

```
-->x=7;if x >= 0 disp("positif"), else disp("négatif"), end

 positif
```

3.1.3 Boucle `for`

La syntaxe de la boucle `for` mérite une explication particulière. En effet, la syntaxe `debut : pas : fin` est exactement celle pour créer une matrice ligne dont les valeurs partent de `debut` pour arriver à `fin` par incrément de `pas`. On peut donc écrire les boucle `for` sous la forme :

```
for cpt=L
```

où `L` est une matrice ligne et où le compteur `cpt` prend successivement toutes les valeurs de `L` (dans l'ordre). Cette remarque est très utile pour réécrire certaines boucles plus simplement.

Un exemple sur une ligne :

```
-->for k=1:5, x=k^3, end;
 x  =

    1.
 x  =

    8.
 x  =

    27.
 x  =

    64.
 x  =

    125.
```

Pour mieux comprendre, prenons l'exemple très simple du calcul d'une factorielle. Si on veut écrire un script pour calculer 10! en utilisant la définition $n! = n \times (n-1)!$ et $1! = 1$, on écrira un fichier avec le code :

fact1.sce

```
1  fact=1;
2  for k=1:1:10 do
3      fact=k*fact
4  end;
5  disp(k);
6  disp(fact)
```

Remarque

- Si le pas est omis, il est pris égal à 1.
- `do` peut être omis en passant à la ligne.
  Il peut aussi être remplacé par `then`, par une virgule ou un point-virgule !

On sauve ce code dans un fichier *fact1.sce* que l'on peut exécuter dans *Scilab* avec la commande *('script.sce')* ou en cliquant sur l'onglet *Fichier* → *Éxécuter* et en choisissant le script enregistré.
On peut également utiliser le raccourci *CTRL+E*.

On peut aussi utiliser la récursivité :

facorielle.sce

```
1  function f=factorielle(n)
2  if n<=1 then
3      f=1;
4  else
5      f=n*factorielle(n-1)
6  end
7  endfunction
8
9  disp(factorielle(4));
10 disp(factorielle(10));
```

3.1.4 Boucle while

```
-->y=0;z=0;while z < 25, z=z+9, y=y+1, end
 z  =

    9.
 y  =

    1.
 z  =

    18.
 y  =

    2.
 z  =

    27.
 y  =

    3.
```

Exemple du calcul de la factorielle à l'aide de la boucle *while* :

fact2.sce

```
1  fact=1;cpt=1;
2  while cpt<11 do
3      fact=fact*cpt
4      cpt=cpt+1
5  end;
6  disp(cpt);
7  disp(fact)
```

3.2 Tests de branchement case

Ces tests n'existent pas sous *Python*. Ils permettent d'effectuer une action en fonction de la valeur d'une variable. Un exemple est plus parlant. En voici un :

cas1.sce

```
1  a=floor(4*rand());
2  select a
3  case 0 then
4      disp("zéro")
5  case 1 then
6      disp("un")
7  case 2 then
8      disp("deux")
9  else
10     disp("trois")
11 end
```

3.3 Les sorties de boucles

Elles sont très utiles pour arrêter le cours d'un processus itératif. À cet effet, on utilisera la commande `break`. Exemple :

```
-->a=2
 a  =

    2.

-->for i=1:100, a = 2*a, ...
-->if a> 10**4 then break, ...
-->end;
 a  =

    4.
 a  =

    8.
 a  =

    16.
 a  =

    32.
 a  =

    64.
 a  =

    128.
 a  =

    256.
 a  =

    512.
 a  =

    1024.
 a  =

    2048.
 a  =

    4096.
 a  =

    8192.
 a  =

    16384.
```

4

Communication

4.1 Entrées et sorties

4.1.1 Pause

La commande `halt()` met le programme en attente d'une frappe (pas trop forte !) sur le clavier. Exemple :

halt1.sce

```
1 disp("appuyez sur une touche pour continuer. Alors ? ");
2 halt();
3 disp("Ok")
```

4.1.2 Entrées

- La commande `input()` interrompt l'exécution du programme et propose un message à l'adresse de l'utilisateur. La réponse est attendue au clavier sous forme de scalaire ou de chaîne de caractères. Le programme continue ensuite. Exemple avec une réponse sous forme de scalaire :

temp1.sce

```
1  temp='o'
2  temp=input("Quelle température fait-il (en Kelvins) : ? ...
3  (%pi pour arrêter) ");
4  while (temp~=%pi) do
5      if(temp<0) then
6          rep="Nous ne vivons pas dans le même univers !";
7      elseif(temp>=0 & temp<290) then
8          rep="Ca caille un peu, non ?";
9      elseif(temp>298) then
10         rep="On crève, ici !";
11     else
12         rep="Mmmmh, je vais rester un peu !";
13     end;
14     disp(rep);
15     disp('---------------------------------');
```

```
16      temp=input("Quelle température fait-il (en Kelvins) : ? ...
17      (%pi pour arrêter) ");
18  end;
```

⚠ Pour affecter une chaîne de caractères, il faut préciser l'option `string` :

```
-->rep=input("Dites oui ou non ! ","string"); disp(rep);
Dites oui ou non ! oui

 oui
```

- La commande `x_dialog()` interrompt l'exécution du programme et propose une boîte de dialogue comportant sous forme de chaîne de caractères :
  - un commentaire
  - des valeurs suggérées

```
-->resulat=x_dialog("allez !",["quoi";"où"]);disp(resulat);

!quoi  !
!      !
!où    !
```

La commande affecte ensuite affectée à *resultat* :
- la ou les valeurs suggérées si "OK" est pressé
- [ ] si Cancel est pressé

Le programme continue ensuite.

- La commande `x_choices()` permet de faire un choix parmi une liste.
- La commande `x_choose()` permet de générer un menu. Exemple :

```
->choix=x_choose(["Papa","Maman","La bonne","Et moi"],"A vous !");

-->disp(choix)

    1.
```

  - le premier argument de la commande `x_choose()` est une matrice de chaînes de caractères contenant les différentes options,
  - le second argument constitue le titre du menu (ici "A vous") qui apparaît, lui, au-dessus des options précédemment citées.

Après avoir cliqué sur une des options ou "Cancel", *choix* prend la valeur du numéro de choix indiqué (ici 0 pour Cancel, 1 pour "Papa", 2 pour "Maman", 3 pour "La bonne" ou bien 4 pour "Et moi".

⇒ **Activité 4. 2**
Décrivez le rôle du script suivant :

choix1.sce

```
choix=-1;
while(choix~=0) do
    choix=x_choose(["Papa","Maman","La bonne","Et moi"],"À vous !
");
    select(choix)
        case 0 reponse="Sortie !"
        case 1 reponse="Encore lui !"
        case 2 reponse="Oh non, pas elle !"
        case 3 reponse="Z''êtes sûr ?"
        case 4 reponse="Bon, j''m''y colle !"
    end;
    disp(reponse)
end;
```

4.1.3 Sauvegarder des résultats

- La commande `save()` permet de sauvegarder des résultats ou fonctions dans des fichiers binaires afin de les réutiliser plus tard.

  ⚠ Ces fichiers binaires ne peuvent être lus que par *scilab*. Pour être lus par d'autres logiciels, il faudra utiliser `write` et `read` : vous pouvez vous reporter au paragraphe 4.2 page 55.

- La commande `load()` permet, elle, de charger le fichier précédemment sauvegardé.
  Exemple :

jour1.sce

```
date1=("Nous sommes le : ");
save("ex1",date1)
disp(date1 + date())
clear; // effacement de la mémoire temporaire
load("ex1");
disp(date1 + date());
```

4.2 Manipulation de fichiers externes

4.2.1 Manipulation de fichiers

On peut aussi fabriquer des matrices de chaînes de caractères dans un fichier avec les commandes `write` ou `mputl` et les lire avec `read` ou *mgetl* :

```
-->M=['anglais' 'hello';
-->'français' 'bonjour';
-->'espagnol' 'hola']
 M  =
```

```
!anglais   hello    !
!                   !
!français  bonjour  !
!                   !
!espagnol  hola     !

-->mputl(M,'essai.txt')


-->mgetl('essai.txt')
 ans  =

! anglais   !
!           !
! français  !
!           !
! espagnol  !
!           !
! hello     !
!           !
! bonjour   !
!           !
! hola      !
```

mputl permet de récrire sur un fichier, ce que ne permet pas `write` :

```
-->mputl(M,'essai.txt')
-->mputl(M,'essai.txt')//réécriture sur le fichier

-->write('essai.txt',M)//réécriture sur le fichier
                    !--error 240
File essai.txt already exists or directory write access denied
```

Les commandes d'ouverture et d'effacement de fichiers existent également :

- `mclose()` permet de fermer un fichier ouvert,
- `mdelete()` efface un ou des fichiers!!!

4.2.2 Manipulation de répertoires

On peut aussi manipuler des répertoires.

Attention aux bêtises, toujours possibles et irréversibles : croyez en l'expérience de votre serviteur...

commande	description
`chdir`	change le répertoire courant de *Scilab*
`createdir`	crée un nouveau répertoire
`dir`	retourne liste des fichiers
`isdir`	demande si l'argument est un répertoire
`ls`	retourne les fichiers du répertoire
`mkdir`	crée un nouveau répertoire
`pwd`	affiche le répertoire courant de *Scilab*
`removedir`	efface un répertoire
`rmdir`	supprime un repertoire

TABLE 4.1 – Manipulation de répertoires

5

Sorties graphiques

Un site extrêmement complet :
http://fr.wikibooks.org/wiki/Découvrir_Scilab/Graphiques_et_sons

5.1 Généralités

Scilab possède une librairie graphique très puissante, qu'il n'est pas possible de présenter entièrement ici. Pour pouvoir débuter avec *Scilab*, il faut commencer par comprendre comment tracer une courbe du plan comme le graphe d'une fonction, par exemple.

Pour tracer une courbe, *Scilab* possède une fonction `plot2d(x,y)` qui trace une ligne brisée en reliant par des segments de droites les points de coordonnées *(x(i),y(i))*. Pour avoir un affichage convenable du graphe, il faut donc avoir suffisamment de points pour que la courbe ait un aspect "lisse" et pas celui d'une ligne brisée. La démarche est donc la suivante :

- découper l'intervalle en un nombre suffisant de valeurs rangées dans une matrice colonne x,
- calculer les valeurs correspondantes de $f(x)$ (par exemple avec `feval`) et les stocker dans une matrice colonne y,
- exécuter la commande `plot2d(x,y)` pour ouvrir la fenêtre dans laquelle s'affiche le graphe.

Il faut donc se donner :

- un vecteur d'abscisses, par exemple à l'aide d'incréments : `t=0:0.1:2*%pi`,
- un vecteur d'ordonnées. On peut le construire à l'aide d'une fonction vecteur. On rappelle qu'en *Scilab* : $f(v) = [f(v1),f(v2),...,f(vn)$ si $v = [v1,v2,...,vn]$.

Remarques

- Si x et y sont des vecteurs, alors *Scilab* trace les points de coordonnées (x_i, y_i).
  Les deux vecteurs doivent avoir la même dimension. Il n'est pas obligatoire que ce soit des vecteurs colonnes, mais il vaut mieux en prendre l'habitude, pour le cas général.
- Si x et y sont des matrices, elles doivent avoir les mêmes dimensions. *Scilab* trace les courbes correspondant à chaque colonne de x et y. La $i^{\text{ème}}$ courbe correspond aux points définis par la $i^{\text{ème}}$ colonne de x en abscisse et la $i^{\text{ème}}$ colonne de y en ordonnée.
- Si x est un vecteur (colonne) et y une matrice, alors *Scilab* trace les courbes correspondant à chaque colonne de y pour ordonnées et x pour vecteur des abscisses. Cela impose bien que x soit un vecteur colonne.

Exemples : les deux suites d'instructions font la même chose :

plot1.sce

```
1  x=linspace(0,3*%pi,100);
2  y=x.*sin(x)
3  xset("window",1) //première fenetre
4  plot(x,y)
5
6  x=linspace(0,3*%pi,100);
7  deff("y=f(x)","y=x.*sin(x)")
8  xset("window",2) //deuxième fenetre
9  fplot2d(x,f)
```

5.2 La commande *plot2d*

Lorsqu'on veut superposer plusieurs courbes avec les mêmes échelles de représentation, il est préférable d'utiliser la commande `plot2d` qui autorise des styles différents pour chaque courbe.
La syntaxe générale est la suivante :

```
plot2d(abscisses, ordonnées, style, cadre, légendes, bornes, graduation)
```

Les deux premiers arguments sont obligatoires, les suivants sont facultatifs.

⚠ Comme il faut les mettre dans l'ordre, cela signifie que si un argument optionnel est précisé, les précédents doivent l'être aussi.

5.3 Fenêtres

commande	description
`clf()`	efface la fenêtre courante
`xbasr()`	redessine une fenêtre graphique
`xclear()`	efface une fenêtre graphique.
`xdel()`	supprime des fenêtres graphiques
`xname()`	change le nom de la fenêtre graphique courante

TABLE 5.1 – Opérations sur les fenêtres graphiques

5.4 Abscisses et ordonnées

Ce sont des matrices de même dimension. Si ce sont des vecteurs (donc une seule courbe à tracer), ils peuvent être en ligne ou en colonne.
S'il y a plusieurs courbes, elles doivent correspondre à autant de colonnes.

Exemple avec une courbe :

```
-->t=(0:0.1:2*%pi);

-->plot2d(t,sin(t))

-->plot2d(t',(sin(t))') // mieux !
```

Le résultat ci-dessous (*courbe1.eps*) :

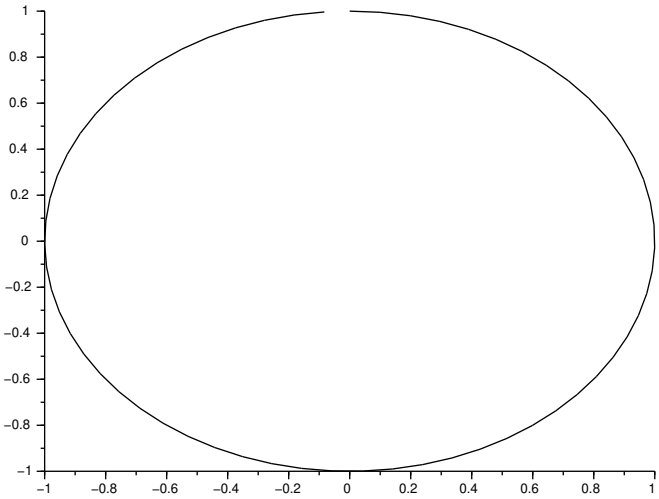


FIGURE 5.1 – 1 courbe

⇒ **Activité 5.3**

Écrivez le programme (*param.sce*) permettant de représenter la courbe paramétrée définie sur l'intervalle $[-1.5, 1.5]$ par :

$$x(t) = 1 + t^2 - t^3$$
$$y(t) = t^2 + t^3 - t^5$$

Le programme :

On obtient le graphe suivant (*param.eps*) :

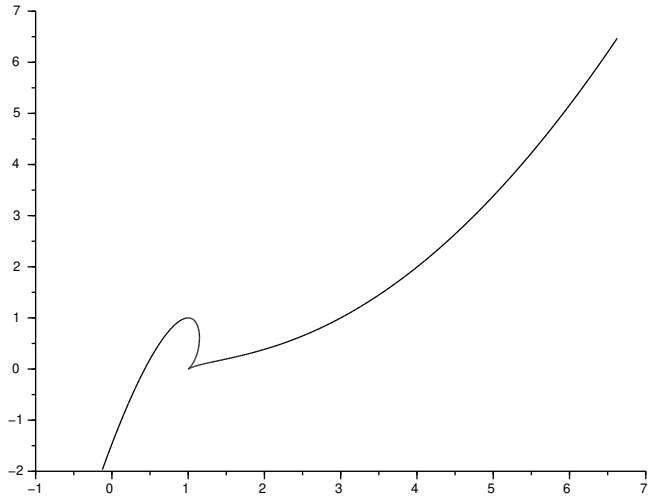


FIGURE 5.2 – Courbe paramétrée

5.5 Style

C'est un vecteur ligne dont la dimension est le nombre de courbes à tracer (nombre de colonnes des matrices *abscisses* et *ordonnées*).
Les coordonnées sont algébriques :

- si le style est positif, les points sont reliés par des segments,
- si le style est nul, les points sont affichés comme des pixels nuls,
- si le style est négatif, des marques de formes particulières sont affichées. On peut aussi préciser un style par une lettre ou un symbole.

Numéro	Style		
1	noir	17	bleu
2	bleu foncé	18	bleu pâle
3	vert clair	19	marron foncé
4	bleu	20	marron
5	vermillon	21	marron clair
6	rose	22	violet foncé
7	jaune	23	violet
8	blanc	24	violet clair
9	bleu	25	brun
10	bleu foncé	26	brun clair
11	bleu	27	brun pâle
12	bleu clair	28	rose foncé
13	olive	29	rose
14	vert	30	rose clair
15	vert clair	31	rose pâle
16	bleu	32	orange

TABLE 5.2 – Couleurs de courbes

Un exemple pour illustrer :

plot11.sce

```
clf()
deff('y=f(x)','y=1-exp(-x/i)')
x=[0:0.01:5]';
for i=1:32 do
y=feval(x,f);
plot2d(x,y,i,axesflag=5,frameflag=6)
end
xgrid(3)//grille verte
```

Le résultat (*plot11.eps*) :

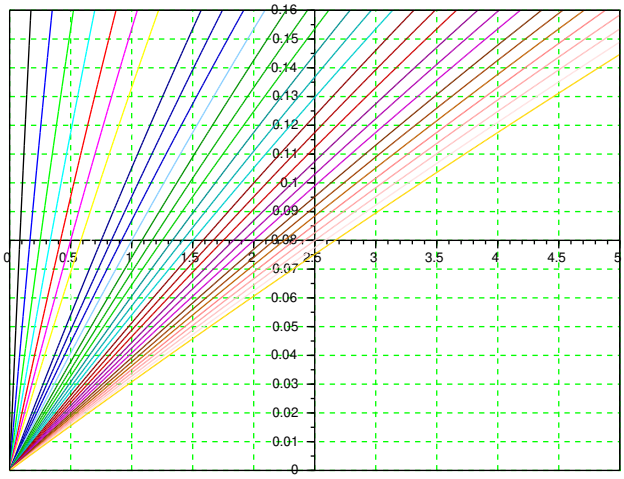


FIGURE 5.3 – Graphe en couleurs

Avec la commande `plot` (et seulement avec cette commande !), il faut utiliser des lettres pour les couleurs ("r" pour rouge, "b" pour bleu, "g" pour vert clair, "k" pour noir, "m" pour rose, "y" pour jaune, ...), par exemple :

```
-->x=linspace(-2*%pi,2*%pi,100)';

-->plot(x,sin(x),style="r");
```

On peut aussi utiliser avec `plot2d` les noms de couleurs prédéfinis, par exemple :

```
-->x=linspace(-2*%pi,2*%pi,100)';

-->plot2d(x,[sin(x),cos(x)],style=[color("red"),color("brown")]);
```

La liste des couleurs peut être consultée ici :
http://help.scilab.org/docs/5.4.0/fr_FR/color_list.html

Un autre exemple coloré :

coul1.sce

```
x = 1:15;
y = 1:15;
map = hsvcolormap(15);
xset('colormap',map);
```

```
5  for n=1:15
6    plot(x(n),y(n),'b-','marker','d','markerfac',map(n,:))
7  end
```

qui donne (*scilab-coul1.eps*) :

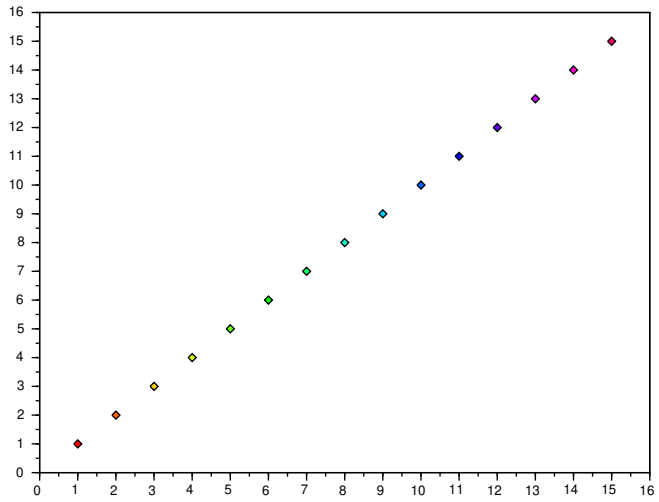


FIGURE 5.4 – Points en couleurs

Numéro	lettre	Style
0	.	•
-1	'+'	+
-2	'x'	x
-3		⊕
-4	'd'	♦
-5		◊
-6		△
-7		▽
-8		♣
-9	'o'	○
-10		∗
-11		□
-12		▷
-13		◁
-14		⋆
	'X-'	points et lignes

TABLE 5.3 – Styles de points

5.6 Cadre

Ce paramètre est une chaîne de caractères formé de trois chiffres.

- le premier code la présence de légendes (0 ou 1),

- le deuxième code le calcul des échelles en abscisse et en ordonnée,
- le troisième code le tracé des axes ou du cadre.

Par défaut, l'argument `cadre` vaut *021* : pas de légende, échelles automatiques, présence des axes.
Si on superpose deux graphiques avec cette option par défaut, les échelles ne seront pas les mêmes.
Si on veut forcer la même échelle, on pourra utiliser l'option de cadre *000* :

plot2.sce

```
1  x=linspace(0,3*%pi,100);
2  y1=x.*sin(x)
3  plot2d(x,y1)
4  y2=2*y1;
5  plot2d(x,y2,1,"000")
```

5.7 Titres et légendes

Ce sont des chaînes de caractères contenant les différentes légendes, séparées par @.

On peut aussi indiquer des options intéressantes en dehors de la fonction `plot2d` :

- Pour placer un titre au milieu d'un graphique : `titlepage('titre')`,
- Pour placer un titre au bas d'un graphique : `xtitle('titre')`.

Ces deux commandes sont à placer après l'ordre `plot2d`.

Exemple avec le programme :

plot3.sce

```
1  x=linspace(0,3*%pi,100);
2  X=x'*ones(1,5);
3  y=x.*sin(x);Y=y'*[1:5];
4  styles=[1:6]
5  legendes="x sin(x) @ 2 x sin(x) @ 3 x sin(x) @ 4 x sin(x) @ 5 x sin(x
   )"
6  plot2d(X,Y,styles,"121",legendes)
```

Le résultat (*scilab-plot3.eps*) :

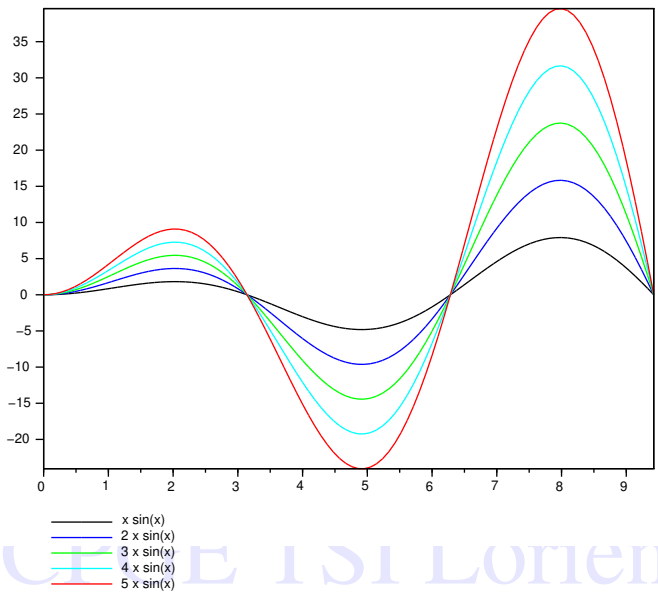


FIGURE 5.5 – Graphe avec titre

Pour améliorer l'aspect du graphe, on pourra utiliser d'autres options de la fonction `plot2d` (voir l'aide en ligne pour plus de détails), par exemple :

- `axesflag` pour gérer le positionnement des axes, (`axesflag=5` pour avoir des axes qui se croisent en (0,0)),
- `frameflag` pour gérer la taille de la fenêtre et les échelles (`frameflag=4` pour une échelle isométrique, `frameflag=6` pour avoir une échelle avec des graduations simples"),
- `xgrid()` pour afficher une grille,
- `legends()` pour afficher une légende.

5.8 Bornes

C'est le rectangle de représentation, décrit par les deux coordonnées du coin inférieur gauche, suivies de celles du coin supérieur droit. Un exemple :

plot4.sce

```
1 x=[-%pi:0.01:%pi]';
2 y=x.*sin(x);
3 plot2d(x,y,4,"011","",[-3,-0.2,3,2])
```

5.9 Graduations

C'est un vecteur de 4 entiers qui permet de préciser la fréquence des graduations et sous-graduations en abscisse et en ordonnée.

plot5.sce

```
1 x=[-%pi:0.01:%pi]';
2 y=x.*sin(x);
3 plot2d(x,y,4,"011","",[-3,-0.2,3,2],[2,6,4,5])
```

Le résultat :

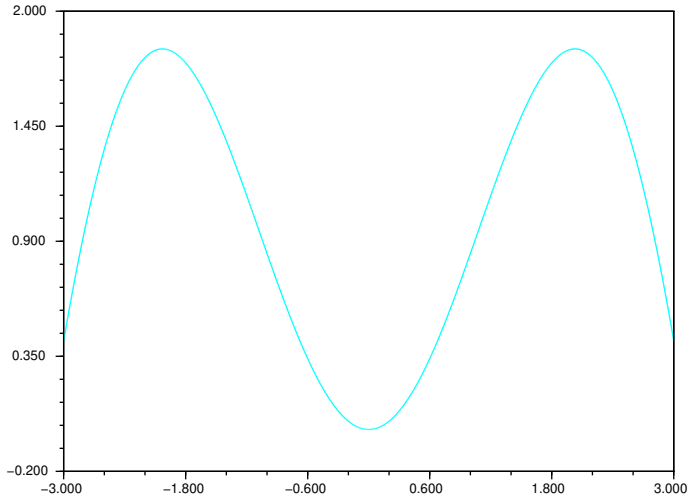


FIGURE 5.6 – Graduations

Une autre façon de procéder :

plot6.sce

```
clf()
deff('y=f(x)','y=x+cos(x)')
x=[-%pi:0.01:%pi]';
y=feval(x,f);
plot2d(x,y,color('red'),axesflag=5,frameflag=6)
plot2d(x,2*y,color('blue'),axesflag=5,frameflag=6)
plot(x, sin(x), "+g", x, cos(x), "-^r")
xgrid(3)//grille verte
```

Le résultat (*plot6.eps*) :

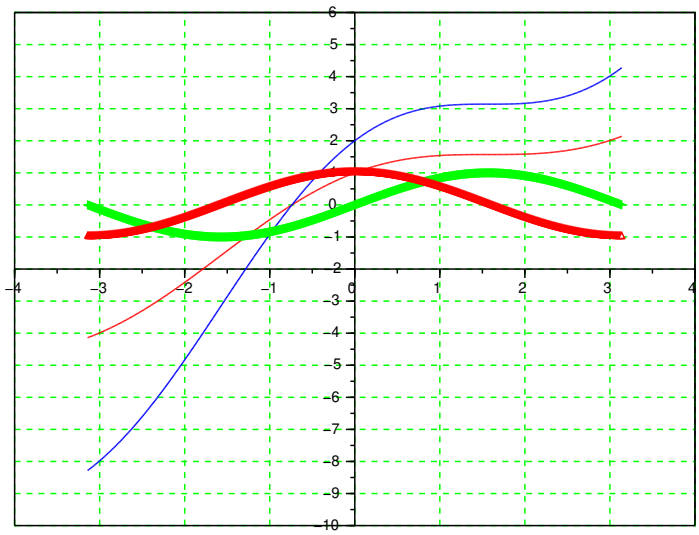


FIGURE 5.7 – Axes et grilles

5.10 Superposition de courbes

Lorsqu'on n'a plus besoin du graphe, on peut détruire la fenêtre ou utiliser la commande `clf()` pour effacer le contenu de la fenêtre. Nous avons déjà vu cela page 59. On peut aussi superposer des courbes en exécutant plusieurs fois `plot2d` sans effacer la fenêtre ou en une seule commande comme ci-dessous :

```
clf()//effacement de la fenêtre
deff('y=g(x)','y=cos(x)/(1+x^2)')
y2=feval(x,g);
plot2d([x x],[y y2],[5 2],axesflag=5,frameflag=6)
xgrid(3)
legends(['f(x)' 'g(x)'],[5 2],2)//affichage d'une légende
```

Le résultat (*graphe2d2.eps*) :

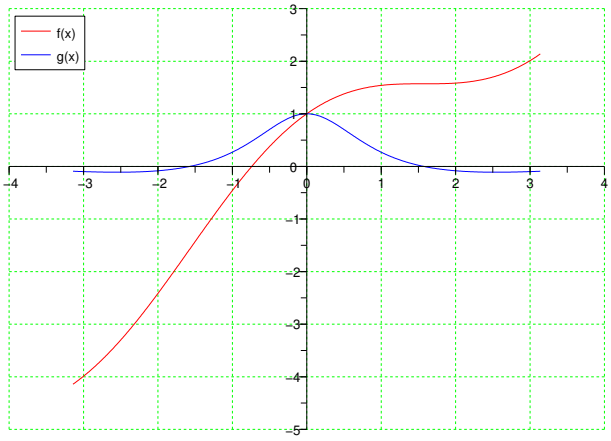


FIGURE 5.8 – Superposition de graphes 1

ou plus simplement :

```
-->clf();x = (0:0.05:2*%pi)';

-->Y = [cos(x), sin(x)];

-->plot(x, Y)
```

Le résultat ci-dessous (*courbe3.eps*) :

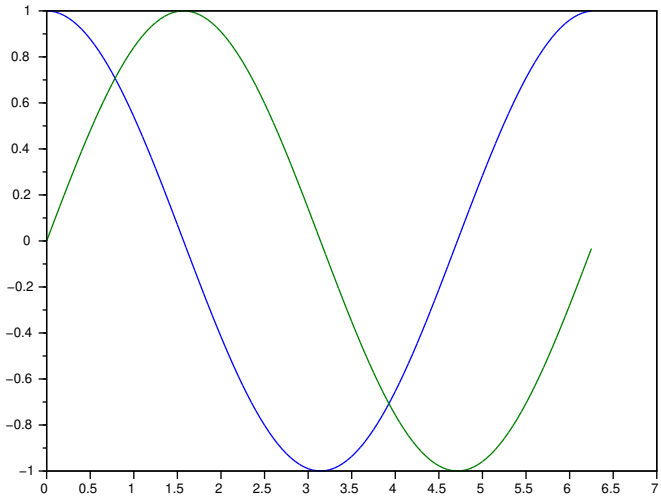


FIGURE 5.9 – Superposition de graphes 2

5.11 Différentes courbes dans des sous-fenêtres

La suite d'instructions suivante :

```
-->subplot(1,2,1)

-->plot2d(x,sin(x),rect=[0,-1,2*%pi,1])
```

```
-->subplot(1,2,2)

-->plot2d(x,cos(x),rect=[0,-1,2*%pi,1])
```

donne (*courbe5.eps*) :

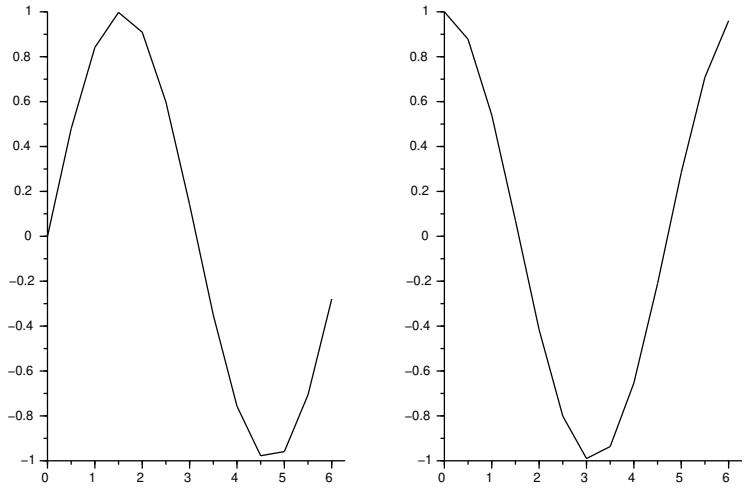


FIGURE 5.10 – Sous-fenêtres

5.12 Différentes courbes dans différents graphiques

On peut afficher plusieurs courbes dans différents graphiques sur la même figure.
Pour pouvoir réaliser cela en *Scilab*, il faut préciser le point d'origine de chaque courbe ainsi que la longueur des axes dans les directions x et y. À cet effet, on utilise la commande `xsetech`.

Exemple : courbes de $f(t) = sin(t)$ et $g(t) = exp(t)$

```
-->t=0:0.1:2*%pi;

-->xsetech([0,0,0.6,0.3],[-1,1,-1,1])

-->xsetech([0,0,0.6,0.3],[-1,1,-1,1]);

-->plot2d(t,sin(t))

-->xsetech([0.5,0.3,0.4,0.6],[-1,1,-1,1]);

-->plot2d(t,exp(t))
```

Le résultat (*courbe4.eps*) :

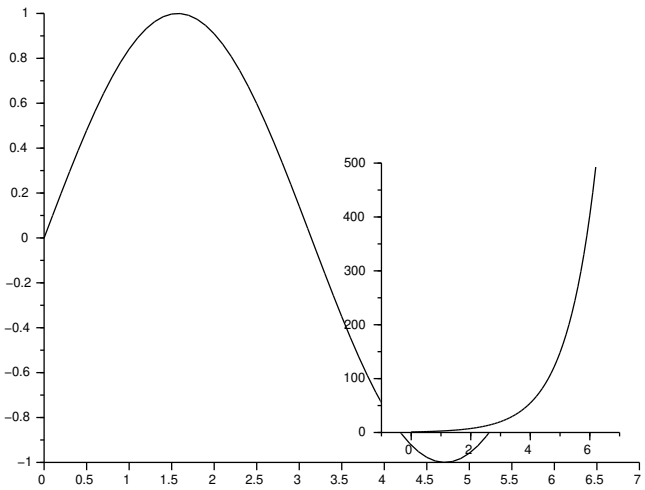


FIGURE 5.11 – 2 courbes dans la fenêtre

5.13 Paramètres de la commande plot2d

La commande plot2d, selon que l'on ajoute à la fin les chiffres 1, 2, 3, 4, ou rien du tout, produit différents types de graphiques.

Commande	type de courbe
plot2d	linéaire par morceaux
plot2d1	idem avec possibilité d'échelle logarithmique
plot2d2	constante par morceaux
plot2d3	barres verticales
plot2d4	flèches

TABLE 5.4 – Paramètres plot2d

Exemple :

plot2dx.sce

```
1  clear; clf();
2  x = [0:0.5:2*%pi]';
3
4  subplot(2,2,1)
5  plot2d1(x,sin(x),rect=[0,-1,2*%pi,1])
6  xtitle("plot2d1")
7
8  subplot(2,2,2)
9  plot2d2(x,sin(x),rect=[0,-1,2*%pi,1])
10 xtitle("plot2d2")
11
12 subplot(2,2,3)
13 plot2d3(x,sin(x),rect=[0,-1,2*%pi,1])
14 xtitle("plot2d3")
15
16 subplot(2,2,4)
```

```
17  plot2d4(x,sin(x),rect=[0,-1,2*%pi,1])
18  xtitle("plot2d4")
```

Le résultat (*courbe2.eps*) :

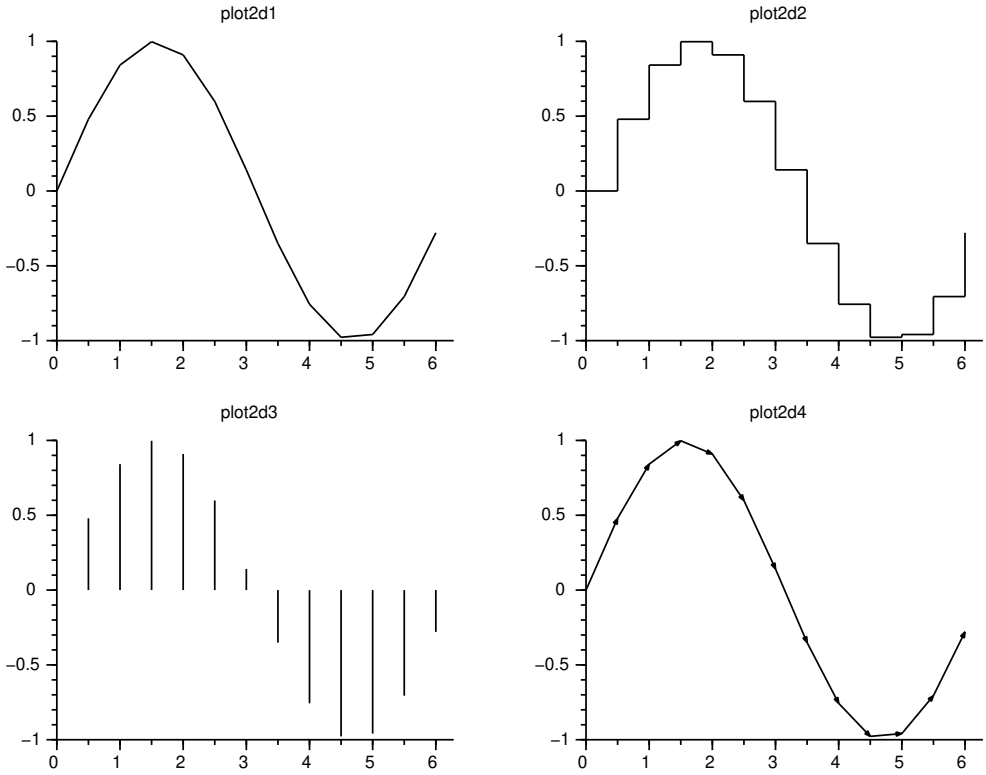


FIGURE 5.12 – Plot2dx

5.14 Objets graphiques

Si on souhaite ajouter des objets sur un graphique, on pourra utiliser :

commande	description
`xarc`	arc d'ellipse
`xfarc`	arc d'ellipse plein
`xarrows`	flèches
`xpoly`	polygone
`xfpoly`	polygone plein
`xrpoly`	polygone régulier
`xpolys`	ensemble de lignes brisées ou de polygones
`xfpolys`	ensemble de polygones pleins
`xrect`	rectangle
`xfrect`	rectangle plein
`xsegs`	ensemble de segments non reliés

TABLE 5.5 – Objets géométriques sur un graphique

5.15 Ajouts de commentaires

Si on souhaite ajouter des commentaires sur un graphique, on pourra utiliser :

commande	description
`xnumb`	nombres
`xstring`	chaîne de caractères (à partir d'un point)
`xstringb`	chaîne de caractères (dans un rectangle)
`xstringl`	chaîne de caractères (dans un rectangle)
`xtitle`	titre du graphique et des axes

TABLE 5.6 – Commentaires sur un graphique

5.16 Graphes particuliers

5.16.1 Dans le plan

commande	description
`histplot`	histogramme
`champ`	champ de vecteurs
`fchamp`	champ de vecteurs défini par une fonction
`grayplot`	surface par rectangles de couleurs
`fgrayplot`	surface par rectangles de couleurs définie par une fonction
`contour2d`	courbe de niveaux
`fcontour2d`	courbe de niveaux définie par une fonction
`polarplot`	courbe en coordonnées polaires

TABLE 5.7 – Représentations particulières en 2D

Un exemple :

polar1.sce

```
1 theta=linspace(0,6*%pi);
2 rho=exp(-theta);
3 polarplot(theta,rho);
```

Le résultat (*polar1.eps*) :

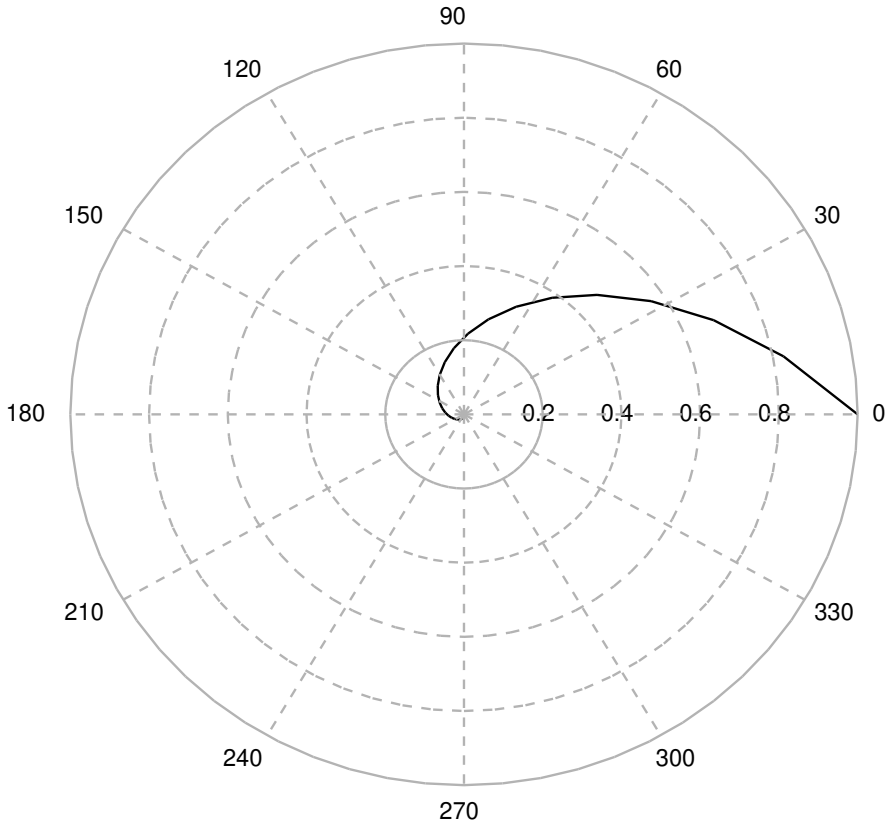


FIGURE 5.13 – Courbe en polaires 1

Un autre exemple :

polar2.sce

```
1  rho=1:0.1:4;theta=(0:0.02:1)*2*%pi;
2  z=30+round(theta'*(1+rho^2));
3  f=gcf();
4  f.color_map= hotcolormap(128);
5  clf();graypolarplot(theta,rho,z)
```

Le résultat (*polar2.eps*) :

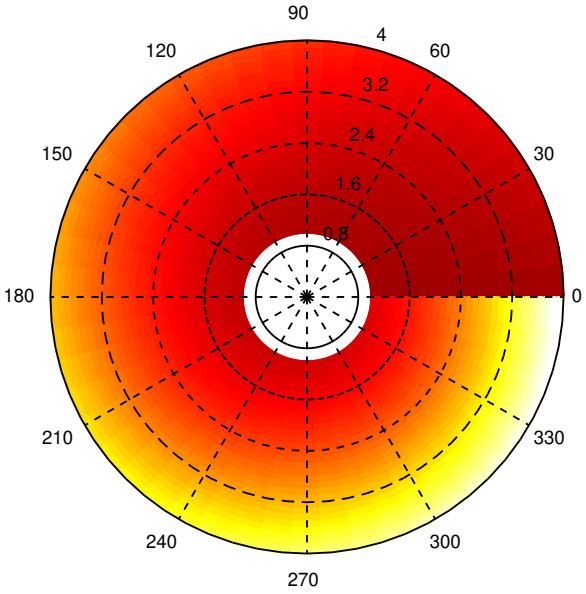


FIGURE 5.14 – Courbe en polaires 2

Encore un si vous insistez :

```
-->x=0:0.1:1; A=x'*x;Fx=cos(8*A); Fy=sin(3*A);champ(x,x,Fx,Fy);
```

Le résultat (*champ0.eps*) :

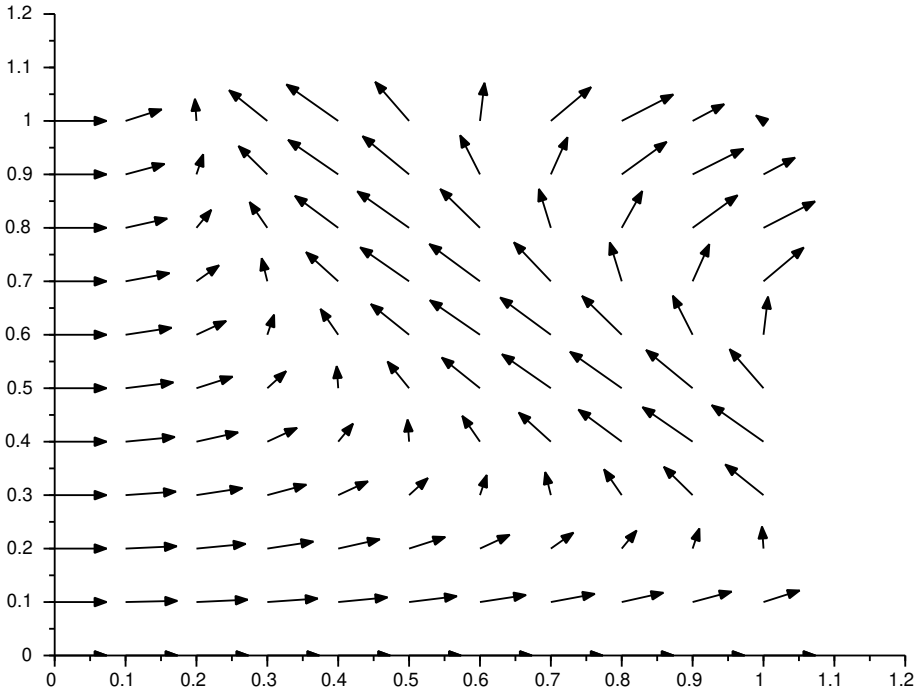


FIGURE 5.15 – Champ 0

5.16.2 Animations

Les 2 programmes suivants réalisent des graphes animés.

Programme :

anim1.sce

```
1  deff('y=f(x,t)','y=t*sin(x)')
2  x=linspace(0,2*%pi,50);theta=0:0.01:1;
3  paramfplot2d(f,x,theta);
```

Le programme correspondant :

champ1.sce

```
1  r = 0.1:0.1:10;
2  w = %pi/10;
3
4  x_i = 0;
5  y_i = 0;
6  for t = 1:length(r),
7  x_f = r(t)*cos(w*t);
8  y_f = r(t)*sin(w*t);
9
10 champ(x_i, y_i, x_f - x_i, y_f - y_i, 0.1);
11
12 x_i = x_f;
13 y_i = y_f;
14
15 sleep(100);
16 end
```

donne :

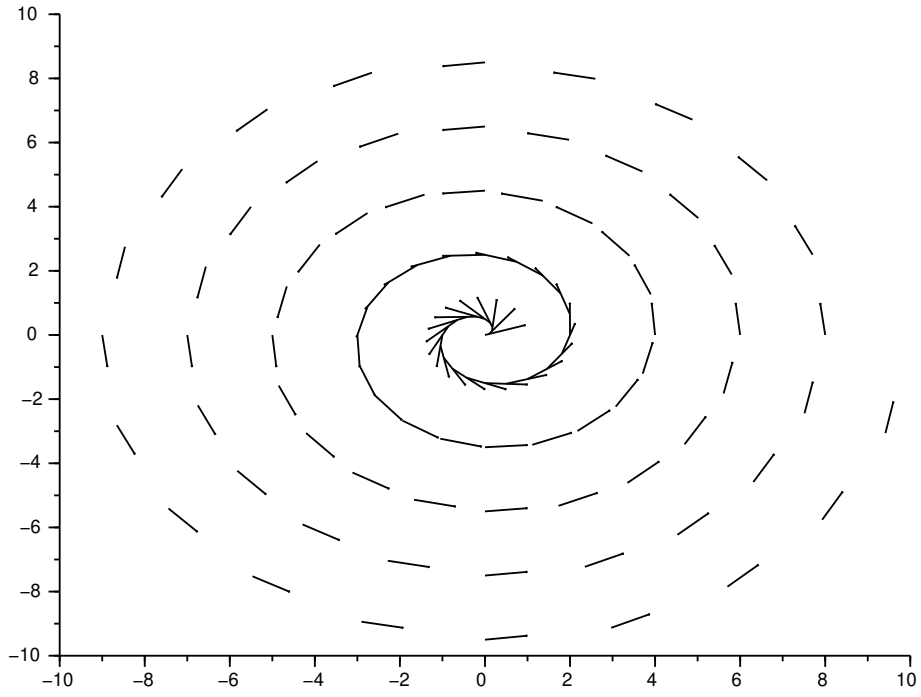


FIGURE 5.16 – Champ 1

5.17 graphes dans l'espace

Scilab offre la possibilité de représenter des graphiques $3D$.
La commande de base est `plot3d(x,y,z)` où x,y et z sont trois matrices.

Pour les fonctions à deux variables, on peut faire des graphiques en trois dimensions avec `plot3d1(x,y,z)`. Il faudra cette fois découper le domaine en petits rectangles (c'est-à-dire découper l'intervalle des x et des y, puis calculer les valeurs correspondantes de $z = f(x,y)$. Pour faciliter l'interprétation du graphe, *Scilab* permet un affichage des niveaux en couleurs à l'aide d'une table des couleurs (colormap en anglais).

Le programme suivant donne un exemple de surface en $3D$:

graphe3d.sce

```
1  clf()
2  deff('z=f(x,y)','r=sqrt(x^2+y^2),z=exp(-r)*cos(3*%pi*r)')
3  x=[-1:0.01:1]';y=x;
4  z=feval(x,y,f);
5  cmap=hotcolormap(64);  //table de 64 couleurs
6  xset("colormap",cmap); //chargement de colormap
7  plot3d1(x,y,z)         //affichage du graphe
```

donne (*graphe3d.eps*) :

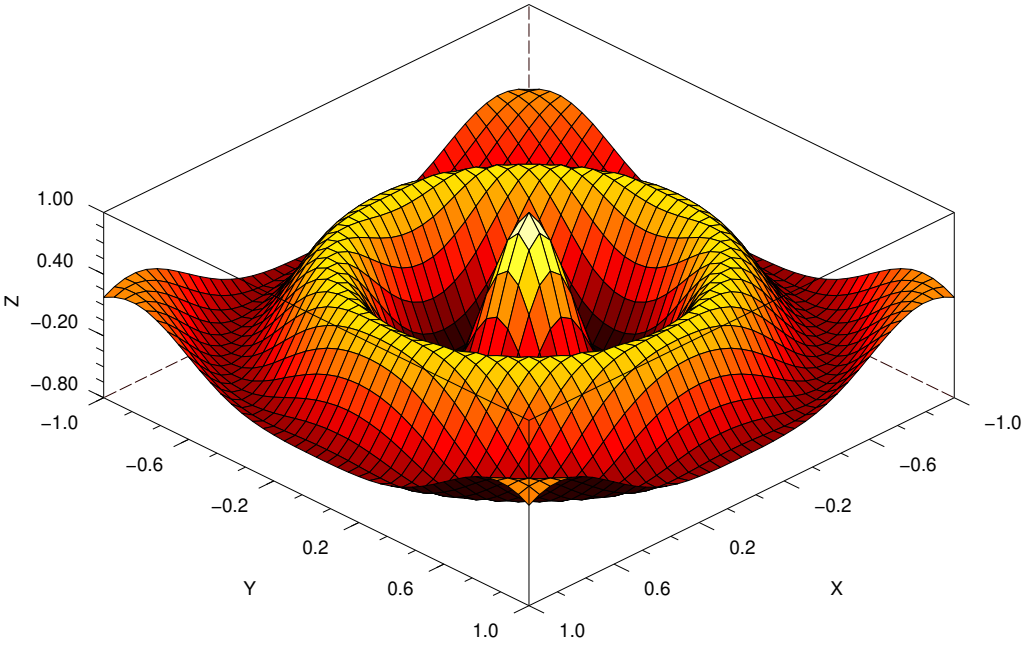


FIGURE 5.17 – Niveaux en couleurs

Voici quelques commandes pour obtenir des graphes dans l'espace :

commande	description
`param3d`	courbe paramétrique
`param3d1`	plusieurs courbes ou points
`plot3d`	surface dans l'espace
`fplot3d`	surface dans l'espace définie par une fonction
`plot3d1`	surface par niveaux de couleurs
`fplot3d1`	surface par niveaux de couleurs définie par une fonction
`eval3dp`	surface paramétrée
`hist3d`	histogrammes

TABLE 5.8 – Représentations particulières en 3D

Un exemple d'utilisation :

helice1.sce

```
1  t=0:0.1:5*%pi;
2  param3d(sin(t),cos(t),t/10,35,45,"X@Y@Z",[2,3])
3
4  e=gce()
5
6  e.foreground=color('red');
7
8  a=gca();
9  a.rotation_angles=[10 70];
```

Le résultat (après avoir fait tourner la figure avec un clic droit : *helice1.eps*) :

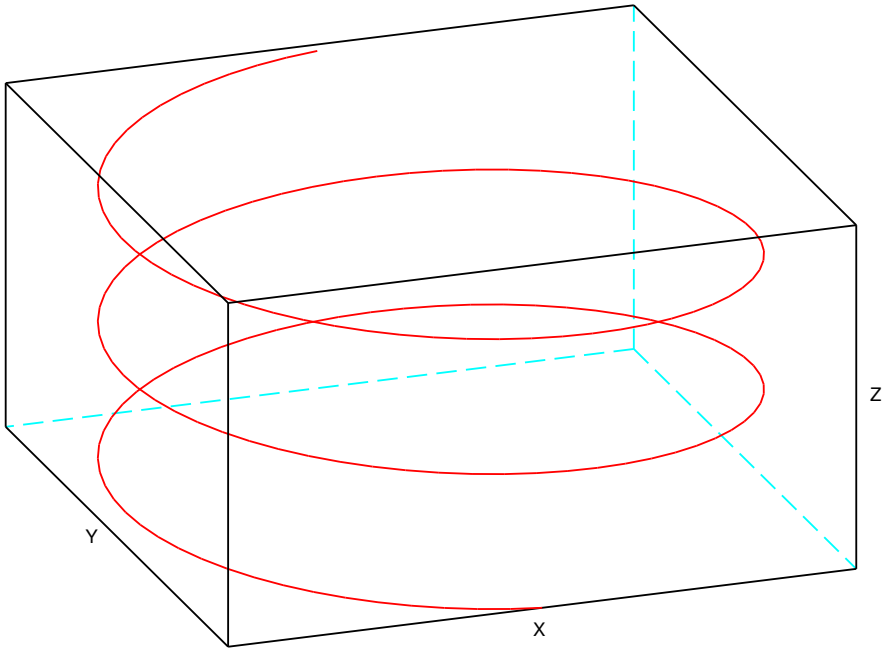


FIGURE 5.18 – Hélice 1

5.18 Enregistrer un graphique

On peut enregistrer ou exporter les graphiques directement depuis la fenêtre.
Une autre solution consiste à l'incorporer dans le script.

- Pour enregistrer un graphique, on utilise la commande :
  `xsave('nom-fichier' [, numéro-fenêtre] )`

- Pour charger le graphique 'nom-fichier' on effectue :
  `xload('numéro-fenêtre' [, numéro-fenêtre] )`

6

Le calcul numérique

Scilab est assez riche en fonctionnalités pour réaliser du traitement numérique. Voici un bref aperçu de ce qu'il possible de réaliser.

6.1 Calcul différentiel, intégration

Vous pouvez visiter cette page :
http://help.scilab.org/docs/5.4.0/fr_FR/section_231e71d1d25567d16b85784757c4950b.html

commande	description
integrate	intégration numérique d'une expression
intg	intégration d'une fonction externe
inttrap	intégration par la méthode des trapèzes
intspline	intégration par approximation par splines
int2d	intégration d'une fonction à 2 variables
int3d	intégration d'une fonction à 3 variables
intc	intégration d'une fonction complexe le long d'un segment
intl	intégration d'une fonction complexe le long d'un arc de cercle

TABLE 6.1 – Calcul différentiel et intégration

Exemple :

integr1.sce

```
1  // fonction carré
2  function [y] = f(x)
3      y = x.^2
4  endfunction
5
6  // intégrale entre 0 et 4
7  Y = integrate("f(x)", "x", 0, 4)
8  disp(Y)
9  // vérification
```

```
10  Y1 = 4^3/3
11  disp(Y1)
```

Autres exemples :

```
-->t=0:0.1:%pi;

-->inttrap(t,sin(t))
 ans  =

    1.9974689

-->intsplin(t,sin(t))
 ans  =

    1.9991349
```

6.2 Traitement du signal

Vous pouvez visiter cette page :
http://help.scilab.org/docs/5.3.1/fr_FR/section_391527fc8412a4b95ddd24555b4b81ab.html

commande	description
dft	transformée de Fourier discrète
fft	transformée de Fourier rapide
convol	produit de convolution
frfit	réponse en fréquence d'un filtre
bode	diagramme de Bode en amplitude et en phase

TABLE 6.2 – Traitement du signal

Exemples :

bode1.sce

```
1  s=poly(0,'s')
2  h=syslin('c',(s^2+2*0.9*10*s+100)/(s^2+2*0.3*10.1*s+102.01))
3  clf();subplot(121)
4  bode(h,0.01,100);
5
6  w=0.01:0.01:2;s=poly(0,'s');
7  G=syslin('c',2*(s^2+0.1*s+2), (s^2+s+1)*(s^2+0.3*s+1));
8  fresp=repfreq(G,w);
9  Gid=frfit(w,fresp,4);
10 frespfit=repfreq(Gid,w);
11 subplot(122)
12 bode(w,[fresp;frespfit])
```

Le résultat (*bode1.eps*) :

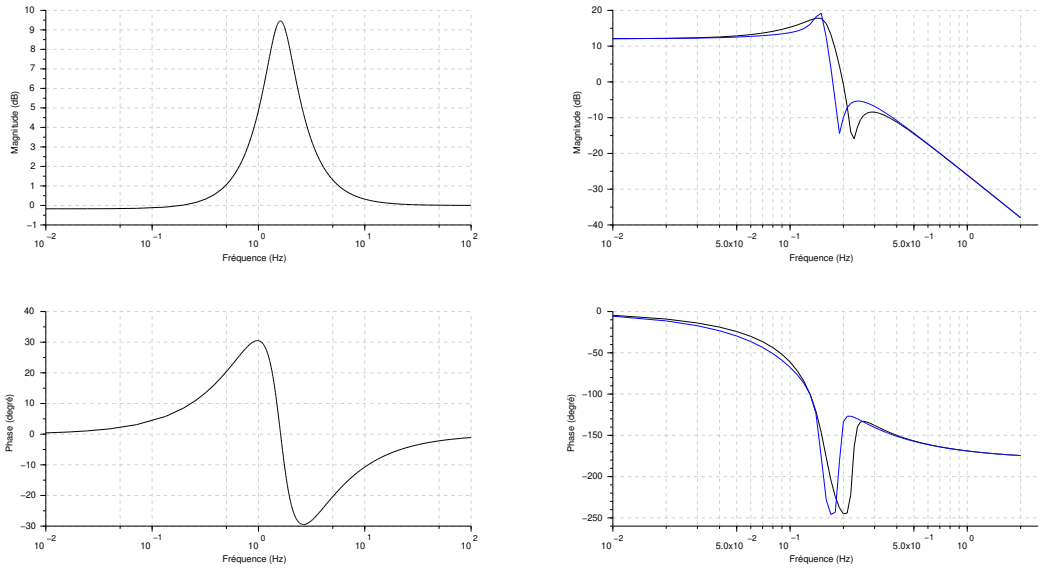


FIGURE 6.1 – Diagramme de Bode

6.3 Optimisation et simulation

Vous pouvez visiter cette page :
http://help.scilab.org/docs/5.4.0/fr_FR/section_e86c83b151e9b4c7c2c10a00273fdb60.html

6.4 Systèmes d'équations

Pour résoudre un système linéaire $Ax = b$, il suffit d'utiliser par exemple la commande `inv(A)*b`. Si on a affaire à des équations non-linéaires, il faut utiliser d'autres méthodes comme celle de Newton.
Scilab possède des commandes permettant d'obtenir directement le résultat (il s'agit d'un résultat approché).

Exemple

Résolution du système :

$$\begin{cases} x^2 = 5 \\ x + 2y = 2 \end{cases}$$

```
-->deff('mafonction=f(x)','mafonction=[x(1)^2-5,x(1)+2*x(2)-2]');

-->fsolve([1,2],f)
 ans  =

    2.236068  - 0.1180340

-->fsolve([12,3],f)
 ans  =

    2.236068  - 0.1180340

-->fsolve([-1,1],f)
 ans  =
```

```
 - 2.236068    2.118034
```

La commande `fsolve`, qui est utilisée pour résoudre un système d'équations non-linéaires, utilise comme on le peut le voir un premier argument qui sert de point de départ à la recherche numérique.
Changer le point de départ peut modifier les valeurs trouvées.

Exemple

Soit le système suivant :

$$\begin{cases} 3x + y &= 5 \\ 4x - y &= 9 \end{cases}$$

On peut le résoudre avec le script suivant :

syst1.sce

```
1  function [y] = f(x)
2      y = [3, 1 ; 4, -1]*x - [5 ; 9];
3  endfunction
4
5  x0 = [0 ; 0];
6  solution = fsolve(x0, f)
7  disp(solution)
```

6.5 Équations différentielles

Les bases :

commande	description
`fsolve`(x_0,f)	résout le système $f(x) = 0$
`ode`(y_0,t_0,T,f)	donne la solution de l'équation différentielle $y'(t) = f(t,y(t)), y(t_0) = y_0$
`champ`(x,y,f_x,f_y)	trace le champ de vecteurs $(f_x(x,y), f_y(x,y))$
`fchamp`(f,t_0,x,y)	trace le champ de vecteurs $(x,y) \longmapsto f((t_0,(x,y)))$

TABLE 6.3 – Équations différentielles

Un exemple simple concernant la charge d'un condensateur :

chargeRC.sce

```
1  // dy/dt=E exp(-t/RC)
2  function ydot=f(t, y),ydot=E*exp(-t/(R*C)),endfunction
3  E = 5;C=0.00005;
4  y0=E;t0=0;t=0:0.01:0.5;
5  for i=1:5
6      R=1000*i
7      y=ode(y0,t0,t,f)
8      plot(t,y)
9  end
```

On obtient (*chargeRC.eps*) :

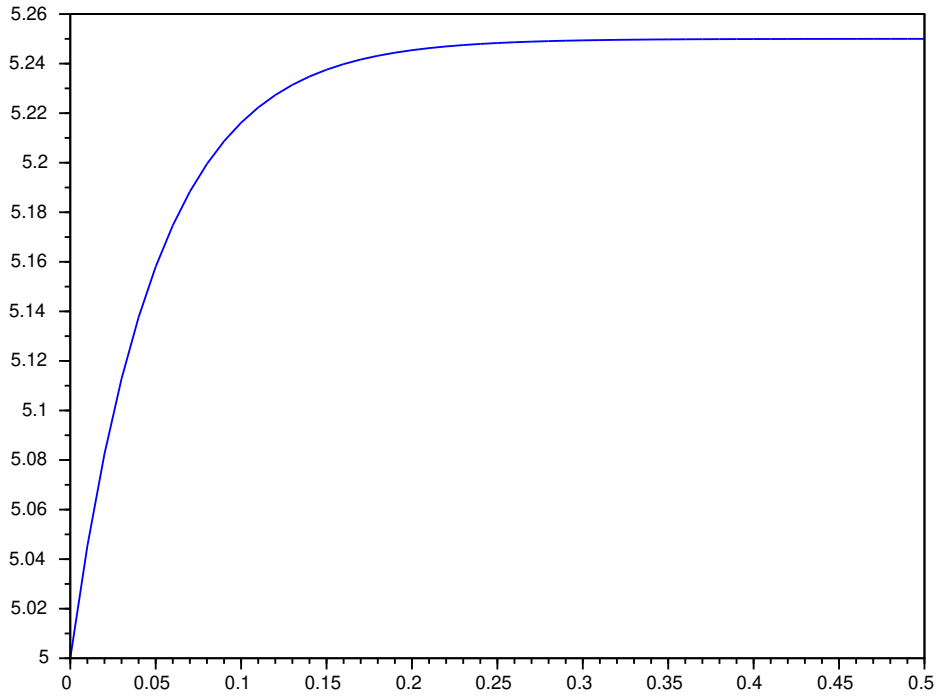


FIGURE 6.2 – Charge d'un condensateur

⇒ **Activité 6.4**

1. En partant du programme *chargeRC.sce*, écrivez un programme permettant de faire varier R de 1000 à 5000 Ω et d'afficher le faisceau de courbes correspondant. On utilisera bien entendu une boucle !
2. Comme vous pouvez le voir, la couleur laisse àn désirer !
   Proposez une mélioration pour avoir de jolies couleurs (vous pourrez remplacer avantageusement `plot`) par `plot2d` et vous inspirer du paragraphe 5.5.

Le programme :

Vous pourrez également tester le programme avec les valeurs 1, 0.1, 1 et 1 par exemple :

RLCSerie.sce

```
1 // Fonction à intégrer : d^2u/dt^2 = -(2/tau)*du/dt - omega0^2*u
2 //Dérivée d'ordre 2 : on procède donc à un changement de variables
3 // f(1) = du/dt
```

```
4  // f(2) = -(R/L)*du/dt - (1/LC)*u
5  function du = RLC(t,u)
6    du(1) =  u(2);                        // du/dt
7    du(2) = -(2/tau)*u(2) - omega2*u(1); // d^2u/dt^2
8  endfunction;
9
10 // Saisie des paramètres de la simulation
11 E = input('Tension de charge du condensateur (en V): ');
12 R = input('Résistance (en ohm - R > 0): ');
13 C = input('Capacité (en farad): ');
14 L = input('Inductance (en henry): ');
15
16 // Définition des conditions initiales de l'intégration
17 t0 = 0 ;                 // origine des temps = 0 s
18 u0 = [E;0];              // la tension initiale est la tension de
   charge du condensateur
19
20 // Autres paramètres de la simulation
21 t1 = 10*2*%pi*sqrt(L*C); // date de la fin de l'expérience, fixée à 1
   0 fois la valeur de 2*pi*sqrt(LC) (en seconde)
22 dt = (t1 - t0)/1000;     // pas de calcul d'intégration
23
24 // Initialisation du vecteur de temps nécessaire à l'intégration
25 t=t0:dt:t1;
26
27 // calcul des paramètres du circuit RLC
28 omega2 = 1/(L*C);   // pulsation propre au carré
29 tau = 2*L/R         // facteur d'amortissement
30 Q = (1/R)*sqrt(L/C) // facteur de qualité
31 mprintf('Pulsation propre: %f - Ammortissement: %f - Facteur de
   qualité du circuit: %f\n',sqrt(omega2),tau,Q);
32
33 // affichage du régime de l'oscillateur
34 if (Q > 0.5) then mprintf('Regime de variation pseudo-periodique\n');
    end;
35 if (Q < 0.5) then mprintf('Regime de variation aperiodique\n'); end;
36 if (Q == 0.5) then mprintf('Regime de variation critique\n'); end;
37
38 // Résolution de l'équation différentielle
39 // i est le vecteur contenant la valeur de l'intensité calculée pour
   chaque pas
40 u = ode(u0,t0,t,RLC);
41
42 // Tracé de la solution
43 clf; // effacement de la fenêtre graphique
44 xtitle('Décharge d''un condensateur dans un circuit RLC','temps (s)',
   'tension (V)');
45 plot2d(t,u(1,:), style= 2); // tracé de la tension aux bornes de C en
    fonction de t
```

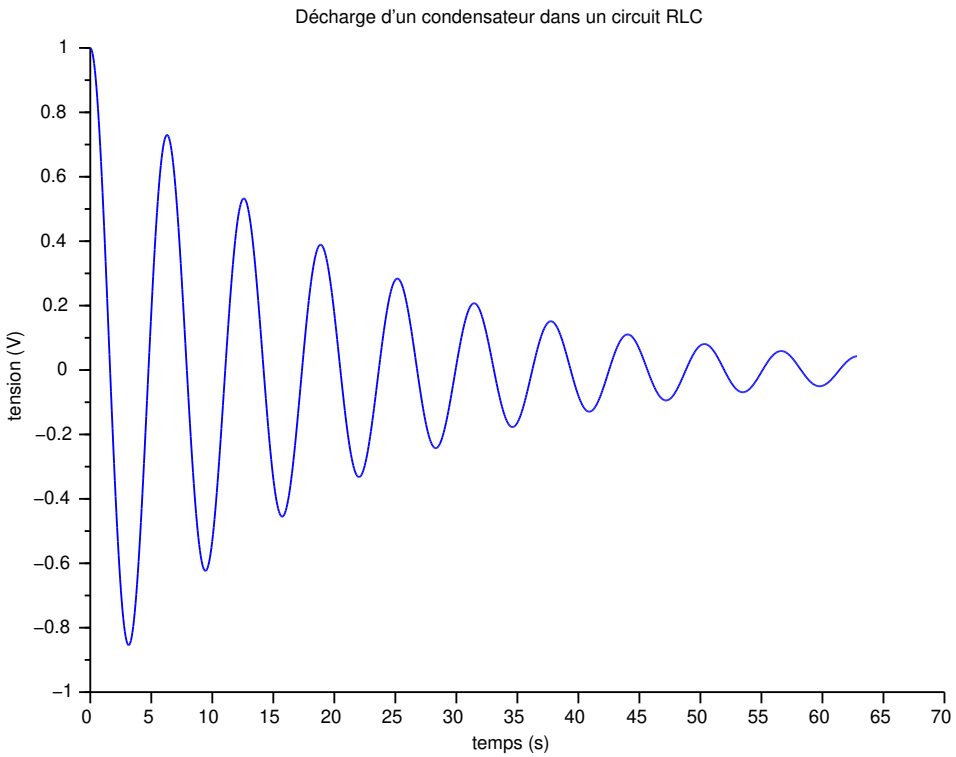


FIGURE 6.3 – Circuit RLC

Exemple

Voici un système d'équations différentielles :

$$\begin{cases} \dfrac{dx}{dt} &= 10(y-x) \\ \dfrac{dy}{dt} &= -xz+20x-y \\ \dfrac{dz}{dt} &= xy-10z \end{cases}$$

```
-->deff('y=h(t,x)','y=[10*(x(2)-x(1)),-x(1)*x(3)+20*x(1)-x(2),x(1)*x(2)-10*x(3) ]
');

-->y=ode([-3;-6;12],0,0:0.01:20,h);

-->param3d(y(1,:),y(2,:),y(3,:));
```

qui donne (*eqdif2.eps*) :

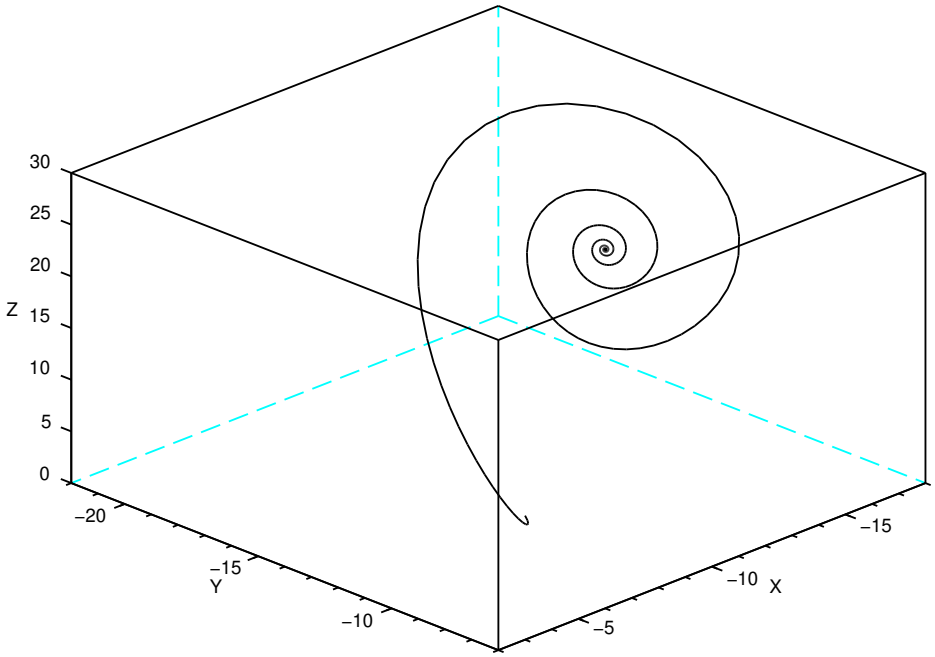


FIGURE 6.4 – Système d'équations différentielles

Pour tracer des champs, on peut utiliser `fchamp(`f`,`t_0`,`x`,`y`)` qui trace le champ de vecteurs associé à l'équation différentielle $y'(t) = f(t, y(t))$ à l'instant t_0.

⚠ La fonction f doit avoir comme variables d'entrée t et y, même si elle ne dépend pas de t ! Voici un exemple où sont superposés le champ de vecteurs et quelques trajectoires de l'équation différentielle :

equa-dif3.sce

```
1  deff('y=f(t,x)','y=[x(2),sin(x(1))-0.5*x(2)]');
2  x=-5:0.8:5;
3  fchamp(f,0,x,x);
4  y=ode([-5;3.5],0,0:0.01:20,f);
5  plot2d(y(1,:),y(2,:));
6  y=ode([-5;3.7],0,0:0.01:20,f);
7  plot2d(y(1,:),y(2,:));
```

qui donne (*eqdif3.eps*) :

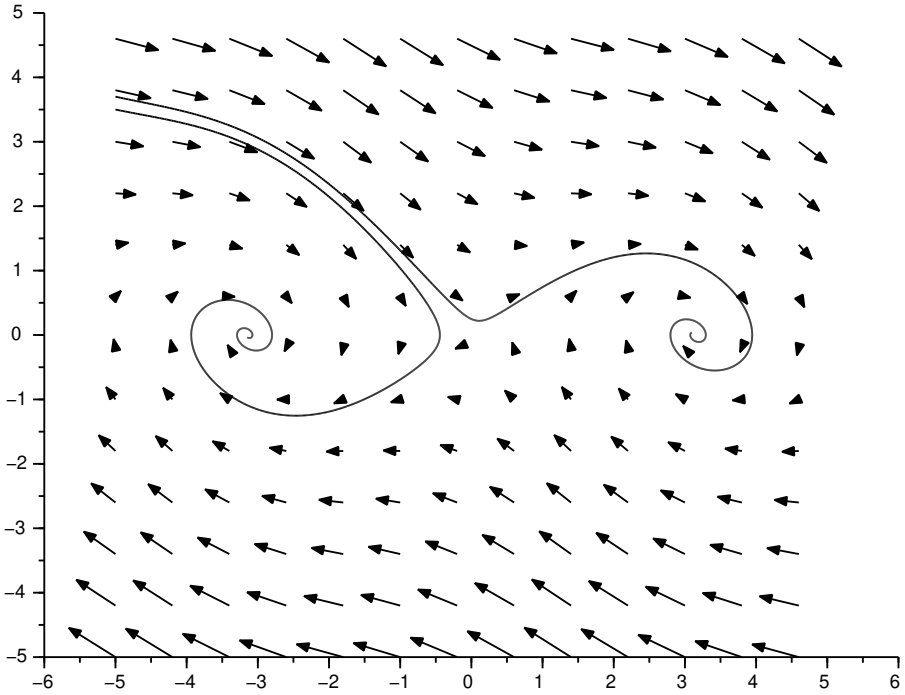


FIGURE 6.5 – Tracé de champ

6.6 Calcul symbolique

Vous pourrez visiter cette page :
http://help.scilab.org/docs/5.4.0/fr_FR/section_8590cf2ca6edfdf3677d9ce7d21be2f1.html

commande	description
addf	addition symbolique
cmb_lin	combinaison linéaire symbolique
ldivf	division à gauche symbolique
mulf	multiplication symbolique
rdivf	division symbolique à droite
solve	solveur symbolique de système linéaire
subf	soustraction symbolique
trianfml	triangularisation symbolique
trisolve	solveur symbolique de système linéaire

TABLE 6.4 – Calcul symbolique

Partie 2

Scilab année 2

7

Le traitement d'images

7.1 Installation du module *sivp*

7.1.1 Sous *Windows*

À partir d'une version récente de *Scilab*, cliquez sur l'onglet *Applications*. Entrez ensuite dans le *Gestionnaire de modules - ATOMS* et cherchez *Image Processing*. Il reste alors à choisir *Scilab Image and Video Processing toolbox* et à l'installer.

7.1.2 Sous *Ubuntu*

Lancez le programme *Scilab*, puis tapez :
`atomsInstall (" makematrix ")`. Suivez ensuite le mode opératoire décrit dans le paragraphe précédent.

7.1.3 Problèmes

- Il se peut que *opencv* soit requis. Sous *Ubuntu*, tapez dans un terminal la commande `sudo apt-get install libcv-dev libcvaux-dev libhighgui-dev`.
- De même avec *libtiff* : sous *Ubuntu*, tapez dans un terminal la commande `sudo apt-get install libtiff4`.
- Même chose avec *libjpeg* : sous *Ubuntu*, tapez dans un terminal la commande `sudo apt-get install libjpeg62`.

Sous *Windows*, pas de souci a priori ...

7.2 Ce que peut lire *sivp*

7.2.1 Extensions supportées

Le module *sivp* peut lire les images et vidéos sous format :

- *bmp*, *dib*

- *jpeg*, *jpg*, *jpe*
- *png*
- *pbm*, *pgm*, *ppm*
- *sr*, *ras*
- *tif*, *tiff*
- *avi*, *mpeg*, ...

7.2.2 Formats supportés

Les formats internes supportés par les fonctions de la bibliothèque sont :

- le format *RVB* $[0:255, 0:255, 0:255]$,
- les images à niveau de gris $G[0:255]$,
- les formats *INT8*, *UINT8*, *INT16*, *UINT16*, *INT32*,
- le format binaire noir et blanc $[0,1]$,
- les formats couleurs usuels *NTSC*, *YCrCb*, *HSV* .

Partie 3

Listes

Table des matières

Figures et tables

Table des figures

Liste des tableaux

Liste des algorithmes

Liste des programmes

Index

Index Scilab

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

V

W

X

Z

Commandes Scilab

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

W

X

Z