MIM 2003-04


**Thierry Clopeau**

Université Lyon 1, 43, Bd. du onze novembre, 69622 Villeurbanne Cedex

**Préambule**

**Scilab** est un environnement dédié à la manipulation numérique et graphique, plus précisément c'est un outil d'ingénierie mathématique qui offre de nombreuses applications en automatisme, algèbre linéaire, traitement du signal, optimisation et résolution de problèmes non-linéaires[1]. C'est un logiciel très similaire au plus connu **Matlab**. La grande différence est que, **Matlab** est un logiciel commercial, tandis que **Scilab** est un logiciel gratuit, “open source”, développé par un institut français : l'**INRIA**[2]. Le côté mercantile de **Matlab** lui confère une interface et un environnement de travail plus confortable. Mais un peu de pratique sous **Scilab**, fait oublier très vite son aspect plus austère. Finalement on se retrouve face à un environnement qui offre des possibilités tout aussi étendues que son concurrent.

**Scilab** est actuellement à sa version 2.7, téléchargeable, entre autre pour les systèmes d'exploitation Linux et Windows (http ://www-rocq.inria.fr/scilab/). Il est possible de consulter sur le site, grand nombre de documentations, de textes divers et contributions. De plus il possède une aide en ligne en français !

Le but de ce manuscrit est de faire découvrir les bases de la pratique et de la programmation de **Scilab** à quelques formations de l'université de Lyon I. Ce texte n'a pas la prétention d'être incontournable, ni exhaustif, mais existe et donc est forcément perfectible. D'ailleurs le texte accueillera avec grande satisfaction toutes remarques et corrections (typographiques et orthographiques incluses).

Les pages suivantes sont organisées comme suit :

**Le 1er Chapitre** est consacré à un rapide tour du propriétaire. Ceci permet de cerner quelques possibilités arithmétiques, algébriques et graphiques offertes, et de s'accommoder avec ce nouvel environnement. Il est recommandé pour les personnes novices à ce genre d'environnement. Le lecteur plus affranchi pourra consulter les tableaux récapitulatifs notamment des variables prédéfinies et fonctions scientifiques usuelles.

**Le second Chapitre** se veut plus technique et nous plonge dans les différents types de variables disponibles (vecteur, matrice, liste ...), avec leurs spécificités de déclaration et manipulation. Des exercices sont proposés pour se familiariser à leur usage.

**Le Chapitre 3** traite des éléments de langage liés à la programmation. En effet **Scilab** est un environnement de travail mais également de programmation. Plus exactement c'est un langage interprété. Seront passés en revue, les éléments de syntaxe de tests conditionnels, boucles et fonctions.

**Le Chapitre 4** permettra de mettre l'accent sur les possibilités graphiques de cet environnement. Quelques exemples d'affichage de courbes 2d et 3d sont donnés.

---

[1] Cette liste n'est pas complète !

[2] Institut Nationale de Recherche en Informatique et en Automatique

# Table des matières

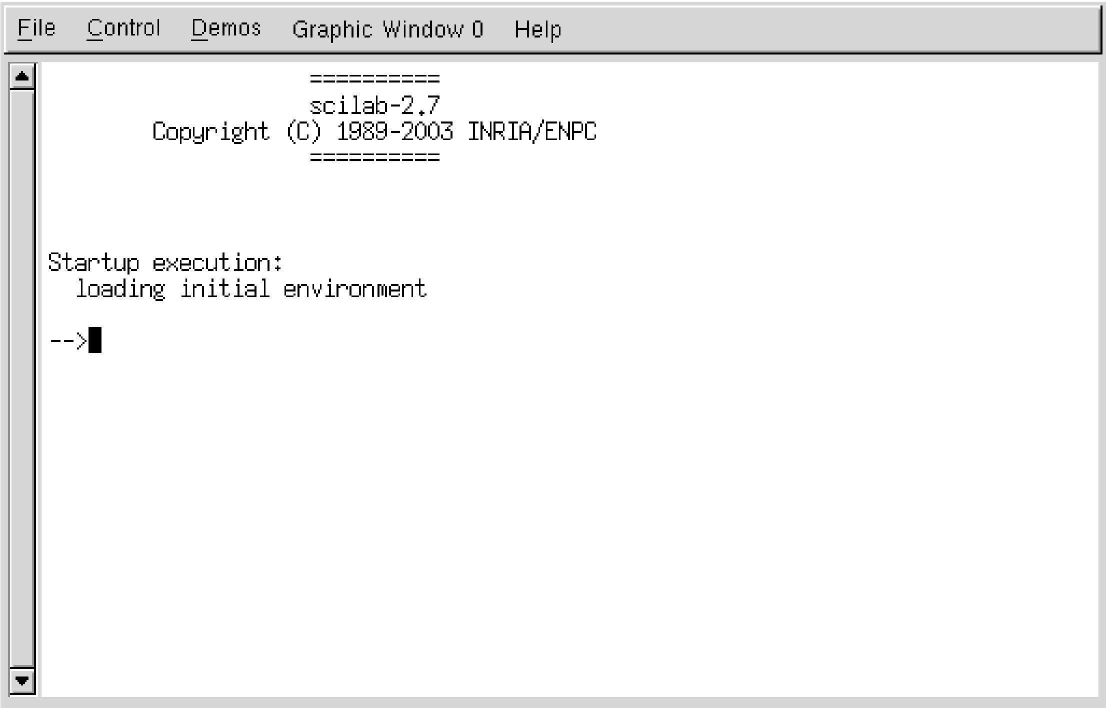
# Chapitre 1

# Prise en main

Dans ce premier chapitre, nous allons passer en revue quelques possibilités basiques de **Scilab**. Le but est de donner au lecteur, un rapide aperçu de cet environnement.

## 1.1 Début de session

Ouvrons une session Scilab : sous shell unix tapez **scilab**.
Doit apparaître la fenêtre :




avec en dernière ligne, le prompt (`-->`) , qui vous invite à taper une commande.

| | |
|---|---|
| + | addition |
| - | soustraction |
| * | multiplication |
| / | division à droite |
| \ | division à gauche |
| ^ ou ** | exponentiation |

FIG. 1.1 – Opérations algébriques

## 1.2 Une calculatrice scientifique

Répondant à l'invitation du prompt on commence par :

```
Fenêtre Scilab
-->2+2
```

après retour chariot nous obtenons

```
Fenêtre Scilab
 ans  =

    4.
```

ce qui nous permet de constater que nous sommes face à une simple calculatrice en ligne, qui évalue l'expression tapée. Les symboles usuels reconnus sont : +, -, * (multiplication), / (division à droite), \(division à gauche), ^ ou ** (exponentiation) et les parenthèses ouvrante et fermente.

La "virgule" des nombres décimaux est remplacée par le ".", et Scilab reconnaît la notation scientifique :

```
Fenêtre Scilab
-->1.2E-1
 ans  =

    0.12
```

ou similairement

```
Fenêtre Scilab
-->1d-4
 ans  =

    0.0001
```

Scilab interprète également la quasi-totalité des fonctions standards telles : `sin`, `cos`, `tan`, `exp`, `log` (logarithme népérien) ... Consultez la table de fonctions usuelles.

| | |
|---|---|
| abs() | Valeur absolue ou module |
| acos() | Cosinus inverse |
| acosh() | Cosinus inverse hyperbolique |
| asin() | Sinus inverse |
| asinh() | Sinus inverse hyperbolique |
| atan() | Tangente inverse |
| atanh() | Tangente inverse hyperbolique |
| ceil() | Partie entière par excès |
| cos() | Cosinus (en radian) |
| cosh() | Cosinus hyperbolique |
| cotg() | Cotangente (en radian) |
| coth() | Cotangente hyperbolique |
| erf() | Fonction erreur : erf(x)=$\frac{1}{\sqrt{2\pi}}\int_0^x e^{-t^2}dt$ |
| erfc() | Fonction erreur complémentaire : erfc(x)=1-erf(x) |
| exp() | Exponentielle |
| fix() | Partie entière la plus proche de 0 |
| floor() | Partie entière inférieure |
| gamma() | Fonction $\Gamma$(x)=$\int_0^\infty t^{x-1}e^{-t}dt$ |
| gammaln() | Logarithme de la fonction $\Gamma$ |
| dlgamma() | Dérivée de la fonction $\Gamma$ |
| log() | Logarithme népérien |
| log10() | Logarithme décimal |
| log2() | Logarithme binaire |
| round() | Arrondi à l'entier le plus proche |
| sign() | Fonction signe |
| sin() | Sinus (en radian) |
| sinh() | Sinus hyperbolique |
| sqrt() | Racine carrée |
| tan() | Tangente (en radian) |
| tanh() | Tangente hyperbolique |

Fig. 1.2 – Fonctions usuelles

| | |
|---|---|
| %pi | $\pi$ |
| %e | $e = 2.7182818$ |
| %i | $i^2 = -1$ |
| %inf | $\infty$ |
| %eps | 1+%eps= 1 |
| %nan | Not A Number |
| %f | false |
| %t | true |
| %io | canal de sortie |
| %s ou %s | monôme polynômial |

FIG. 1.3 – Quelques variables prédéfinies

Fenêtre Scilab
```
-->sin(2* %pi +1)^2 / (tan(10.2) +1)
 ans  =

    0.4295621
```

Noter l'usage de %pi (valeur $\pi$ ), le % est réservé aux variables pré-définies telles : %i la racine carrée de -1, %e=2.7182818 ou encore %eps=4.441E-16 précision machine... voir la table des variables réservées.

## 1.3 Nombres complexes

La variable %i nous permet de composer avec les complexes

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

  - 2.i
```

Les fonctions usuelles sont étendues aux valeurs complexes.

Fenêtre Scilab
```
-->sin(1+%i)
 ans  =

    1.2984576 + 0.6349639i
```

De ce fait Scilab traite implicitement les valeurs réelles et complexes.

**A faire :** évaluer
- imag(1-%i)

- `real(1-%i)`
- `abs(1-%i)`
- `atan(1,-1)`

## 1.4 Format d'affichage

**Scilab** est un logiciel de calcul à précision finie (en principe de l'ordre de 16 chiffres significatifs : "double précision"). Cela fait plus de décimales que les résultats obtenus précédemment. Mais il est possible de fixer ce nombre de décimales à afficher, ainsi que la forme (format scientifique), c'est la commande **format** qui est d'usage :

Fenêtre Scilab
```
-->1/3 , format("v",16); 1/3
 ans  =

    0.3333333
 ans  =

    0.3333333333333
```

ou encore

Fenêtre Scilab
```
-->1/3 , format("e"); 1/3
 ans  =

    0.3333333
 ans  =

    3.333E-01
```

**format** spécifie le type d'affichage (`''v''` ou `''e''`) et le nombre de caractères du résultats (1 caractère pour le signe). Néanmoins il est possible d'afficher plus de décimales que la précision machine `%eps` (qui est la plus grande quantité telle que 1=1+`%eps`), ceci est dépendant en partie de l'unité arithmétique de la machine.

Le mode par défaut est : `format('v',10)`.

**Remarque :** la virgule et le point virgule séparent les instructions, la virgule autorise l'affichage de l'évaluation de l'expression, mais pas le point virgule (Taper `1/3` puis `1/3`;).

## 1.5 Matrices

Étendons un peu plus les capacités de cette super calculatrice, notamment au calcul matriciel pour cela faisons

```
Fenêtre Scilab
-->[1 2 ; 3,4]
 ans  =

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

ce qui rend la composition de matrices aisée, avec une écriture ligne par ligne. Les espaces ou virgule (" " ou ",") jouent le rôle de "séparateur" des colonnes et le ";" celui des lignes, le tout englobé dans des crochets ouvrant-fermant `[ ]`. Pour des raisons pratiques, on peut être amené à écrire la matrice sur plusieurs lignes, alors cette fois c'est le passage à la ligne qui fait office de délimiteur de fin de ligne (de la matrice).

```
Fenêtre Scilab
-->[1 2
--> 3,4]
 ans  =

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

**Remarque :** La matrice peut être réelle ou complexe exemple :`[1+2*%i , %i ; 1 2]`.

La multiplication matricielle devient une simple opération en ligne

```
Fenêtre Scilab
-->[1 2 ; 3 4]*[0 1; 1 0]
 ans  =

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

ou encore

```
Fenêtre Scilab
-->2*[1 2 ; 3 4]
 ans  =

!   2.    4. !
!   6.    8. !
```

Bien sûr quand cela est possible !

```
Fenêtre Scilab
-->[1 ; 2]*[0 1; 1 0]
                   !--error    10
inconsistent multiplication
```

mais

Fenêtre Scilab
```
-->[0 1; 1 0]*[1 ; 2]
 ans  =

!   2. !
!   1. !
```

**A faire :** Taper `[1 2]+1`.

## 1.6 Opérations sur les matrices

Les combinaisons algébriques de matrice engendrent rapidement des opérations admissibles plus nombreuses que le cas scalaire, par exemple la transposition, multiplication terme à terme, multiplication de Kronecker, division à gauche, à droite, terme à terme ... Toutes ces règles demeurant applicables pour le cas scalaire.

Bien sûr les opérations `+`,`-` et `*` fonctionnent (à condition que les tailles soient compatibles).

Faisons un petit tour des opérations matricielles courantes.

**La transposition :** signe ' (simple quotte ou apostrophe)

Fenêtre Scilab
```
-->a=[1 2]'
 a  =

!   1. !
!   2. !
```

**La multiplication terme à terme :** signe `.*`

Fenêtre Scilab
```
-->[1 2].*[2 3]
 ans  =

!   2.    6. !
```

**La division terme à terme :** signe `./`

Fenêtre Scilab
```
-->[1 2]./[2 3]
 ans  =

!   0.5    0.6666667 !
```

**L'exponentiation :** Pour le signe ^ nous avons 2 comportements :

| | |
|---|---|
| ' | transposition |
| + | addition |
| - | soustraction |
| * | multiplication |
| / | division à droite |
| \ | division à gauche |
| ^ ou ** | exponentiation |
| .* | multiplication élément par élément |
| ./ | division à droite élément par élément |
| .\ | division à gauche élément par élément |
| .^ | exponentiation élément par élément |

FIG. 1.4 – Opérateurs algébriques matriciels

Fenêtre Scilab
```
-->[1 2]^2
 ans  =

!   1.     4. !
```

mais

Fenêtre Scilab
```
-->[1 2;3 4]^2
 ans  =

!   7.      10. !
!   15.     22. !
```

alors que

Fenêtre Scilab
```
-->[1 2;3 4].^2
 ans  =

!   1.     4.   !
!   9.     16.  !
```

Autrement dit, si la matrice n'est pas carrée l'exponentiation agit terme à terme, et dans le cas d'une matrice carrée le signe ^ correspond à l'exponentielle de l'opérateur linéaire de la matrice (définie sous forme de série).

Ce dernier exemple illustre l'usage du point (.) devant l'opérateur, cette extension indique que l'opération voulue à lieu élément par élément.

Finissons notre courte description sur les opérations matricielles en remarquant que les fonctions usuelles (`sin`, `cos`, `tan` ...) s'appliquent à chaque terme de la matrice (ou vecteur !)

| | |
|---|---|
| acoshm() | Cosinus inverse hyperbolique matriciel |
| acosm() | Cosinus inverse matriciel |
| asinhh() | Sinus inverse hyperbolique matriciel |
| asinm() | Sinus inverse matriciel |
| atanhm() | Tangente inverse hyperbolique matricielle |
| atanm() | Tangente inverse matricielle |
| cosh() | Cosinus hyperbolique matriciel |
| cosm() | Cosinus matriciel |
| cothm() | Cotangente hyperbolique matricielle |
| logm() | Logarithme matriciel |
| signm() | Fonction signe matriciel |
| sinhm() | Sinus hyperbolique matriciel |
| sinm() | Sinus matriciel |
| sqrtm() | Racine carrée matricielle |
| tanhm() | Tangente hyperbolique matricielle |
| tanm() | Tangente matricielle |

FIG. 1.5 – Fonctions usuelles matricielles

Fenêtre Scilab
```
-->sin([%pi, %pi/2])
 ans  =

!   1.225E-16    1. !
```

**Scilab** possède également des fonctions propres aux matrices carrées[1] (définition sous forme de série) comme : **expm** exponentielle matricielle, avec l'extension "m" pour les différencier.

Un environnement matriciel sans opérations d'algèbre linéaire serait sans intérêt. **Scilab** met à disposition un grand nombre de fonctions telles **inv** (inverse de matrice), **det** (déterminant), **spec** (extraction de valeurs et vecteurs propres), **lu** (décomposition LU) ...

## 1.7 Booléens

Pour compléter cette première description des capacités calculatoires de Scilab, il faut mentionner le calcul booléen. Pour cela, il existe deux variables booléennes `%t` (pour "true") et `%f` (pour "false") qui peuvent être utilisées avec les conjonctions :

– de négation (not)

Fenêtre Scilab
```
-->~%t
 ans  =
```

[1] On vient de voir le comportement de ^ dans le cas d'une matrice carrée (puissance de la matrice).

```
 F
```

– le “et” (and)

```
Fenêtre Scilab
-->%t & %f
 ans  =

  F
```

– le “ou” (or)

```
Fenêtre Scilab
-->%t | %f
 ans  =

  T
```

Scilab sait évaluer les expressions de comparaison du type : `1==2`, `1<2`, `1<=2`, `1>2` et `1>=2`, le résultat est de type booléen.

**Remarque :** Il y a une distinction nette entre le “égale “ d'affectation (`=`) et celui de la comparaison (`==`).

**A faire :** Taper

– `typeof(1==2)`
– `%T`
– `%F`
– `%t|%f&%f`

Enfin le calcul booléen s'étend aux expressions matricielles

```
Fenêtre Scilab
-->[%t %f 1==2]
 ans  =

! T F F !
```

avec

```
Fenêtre Scilab
-->~[%t %f 1==2]
 ans  =

! F T T !
```

mais aussi

```
Fenêtre Scilab
-->[1 2 3]==[3 2 1]
 ans  =

! F T F !
```

les opérateurs & et | agissent sur des matrices de même taille ou alors sur des opérations de type scalaire fois une matrice

Fenêtre Scilab
```
-->%t & [%t %f]
 ans  =

! T F !
```

**A faire :** Taper `typeof([%t %f 1==2])`, `size([%t %f 1==2])`.

## 1.8 Miscelaneous

**Scilab** possède une panoplie quasi complète de types de variables tels les chaînes de caractères, polynômes, fractions rationnelles, fonctions, lists, mlist... De plus l'utilisateur à la possibilité d'en créer de nouvelles (mlist) ainsi que de définir pour ces nouveaux types (ou objets) les opérations usuelles +,-,*,/ ... (surcharge d'opérateur).

**Scilab** intègre un grand nombre de fonctions de l'algèbre linéaire (déterminant, inverse de matrice, valeurs et vecteurs propres ...) ainsi que des procédures de tri et autre indexation. Pour une description plus complète il faut signaler une grand pan (passé sous silence dans ce manuscrit) dédié au traitement du signal.

**Scilab** offre également la possibilité de manipuler des fenêtres de dialogue, de configurer des menus... Tout ce qu'il faut pour développer des applications orienté utilisateur.

Une grande force de cet espace de travail est de donner à tout moment la possibilité de stocker ou d'affecter un résultat à une variable[2].

Fenêtre Scilab
```
-->x=0:0.1:0.8;

-->x
 x  =

!   0.    0.1    0.2    0.3    0.4    0.5    0.6    0.7    0.8  !
```

**Scilab** possède également une riche bibliothèque de fonctions graphiques 2D et 3D.

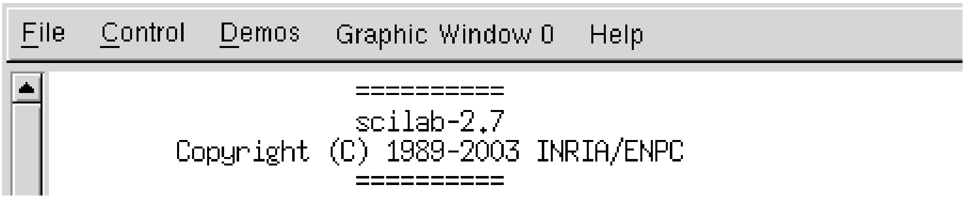
Fenêtre Scilab
```
-->plot(x,sin(x))
```

et bien d'autres.

**A faire :** Taper `plot()`, `plot2d()`, `plot3d()`.

[2]Plusieurs milliers de variables distinctes possibles.

Pour se rendre compte des capacités offertes par cet environnement cliquer sur Demos dans la barre des menus de la fenêtre principale.




## 1.9 Aide

Avant de se lancer dans une description plus complète de l'utilisation de Scilab, finissons cette section par quelques incontournables de tout langage : l'aide en ligne.

Nous avons deux fonctions utiles tout d'abord la fonction **help**

Fenêtre Scilab
```
-->help sin
```

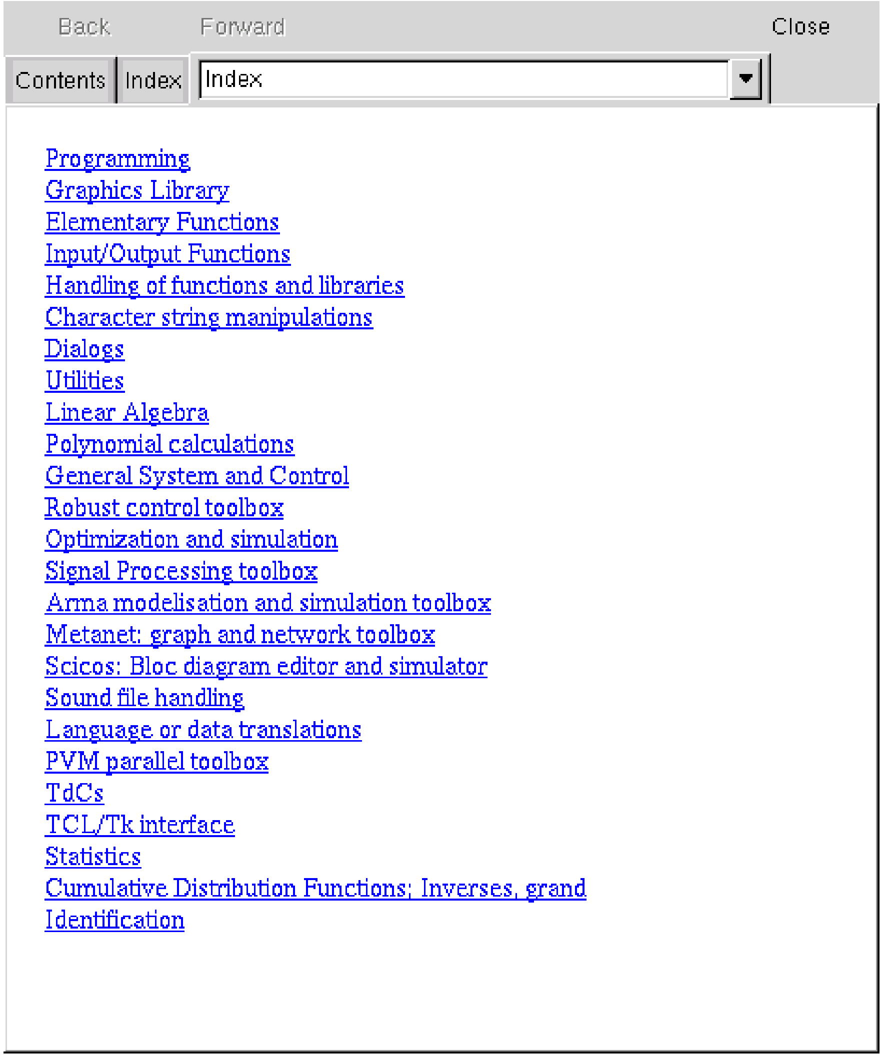
qui renvoie dans une fenêtre un manuel de la commande.

Ensuite la fonction **apropos**

Fenêtre Scilab
```
-->apropos title
```

qui renvoie sur la liste des manuels contenant la chaîne de caractère (title dans l'exemple).

Bien sûr, il est vivement conseillé à l'utilisateur de cliquer sur Help de la barre des menus pour faire apparaître une fenêtre avec un classement thématique. Un clic dessus retourne le help correspondant.




## 1.10 Astuces

La pratique intensive nécessite l'usage de quelques “raccourcis clavier” :

– le première “astuce” est l'utilisation des flèches $\boxed{\uparrow}$ et $\boxed{\downarrow}$ qui permettent de naviguer dans l'historique des commandes déjà exécutées (pour les puriste d'unix il est possible d'utiliser Contrôle-N et Contrôle-P).
– la seconde est le rappel d'une commande avec le point d'exclamation suivi des premières lettres de cette commande :

Fenêtre Scilab
```
-->plot()
-->!p
```

– la troisième n'est pas une astuce Scilab mais sous linux on peut faire du copier-coller avec la souris : en cliquant du bouton gauche on sur-ligne la partie à copier, puis dans la fenêtre Scilab on clique sur le bouton du milieu (coller).

## 1.11 Conclusion

En conclusion de ces premiers pas en **Scilab**, celui-ci se présente comme un logiciel puissant, capable d'évaluer un très grand nombre de fonctions mathématiques. De plus on va voir qu'il peut être utilisé comme un langage de programmation, langage qui sera interprété et non pas compilé comme des langages “classiques” (C/C++, fortran ...).

Scilab est un logiciel interactif et en même temps un logiciel de programmation avec son propre langage. Nous allons dans la suite décrire un certain nombre de type pré-définis ainsi que les règles de manipulation.

## 1.12 Exercices

1. Afficher $\pi$ avec 14 chiffres après la virgule.
2. Calculer $2,5 \times 10^3 + 5$.
3. Écrire le vecteur `(1 2 4)`, obtenir sa transposée.
4. Écrire le vecteur $\begin{pmatrix} 1 \\ i \end{pmatrix}$
5. Calculer le module est l'argument de $\frac{1}{2-i}$.
6. Entrer les matrices :

$$A = \begin{pmatrix} 1 & 2 & 2 \\ 0 & 1 & 2 \\ 0 & 0 & 1 \end{pmatrix} \quad B = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

   Faire le produit matriciel $A \times B$, puis multiplier terme à terme $A$ et la transposée de $B$.

# Chapitre 2

# Variables et types de variables

Ce chapitre est sûrement la partie la plus fondamentale que tout utilisateur de Scilab doit maîtriser. Il définit les opérations de création et de manipulation d'objets multi-indicés que peuvent être les matrices (de réels, booléens ...).

## 2.1 Variable

### 2.1.1 Définition

Le signe "=" est le signe d'affectation pour une variable. L'affectation n'est pas explicitement déclarée au préalable (par son type et/ou sa taille) mais déterminée à l'écriture :

Fenêtre Scilab

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

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

**Remarque 1 :** Les variables sont formées d'une chaîne alphanumérique commençant par une lettre (mais aussi `%` `$` `?` `#`) de 24 caractères (au delà ils ne sont pas pris en compte).
**Remarque 2 :** Il n'est pas possible de réaffecter des variables pré-définies ainsi que les fonctions primitives[1].

Fenêtre Scilab

```
-->%e=1
     !--error    13
redefining permanent variable
-->exp=2e-3
   !--error    25
bad call to primitive :exp
```

[1]Il est également possible de protéger des variables voir **predef**

Une fois la variable définie pour voir son contenu il suffit de taper son nom et de valider :

Fenêtre Scilab
```
-->A
 A  =

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

**Remarque 3 :** Lors de l'exécution d'une commande apparaît la réponse **ans**, c'est une variable qui peut être réaffectée.

Fenêtre Scilab
```
-->[2 3]
 ans  =

!   2.    3. !

-->B=ans
 B  =

!   2.    3. !
```

**Remarque 4 :** Si une variable existe et contient par exemple une matrice, l'affectation de cette variable à une autre quantité (pas forcément de même type) n'engendre pas de message d'erreur ni avertissement.

### 2.1.2 Gestion

Il existe une commande qui permet de tester si une variable existe déjà

Fenêtre Scilab
```
-->isdef('A')
 ans  =

  T
```

noter le passage en argument entre apostrophes (chaîne de caractères) et la réponse sous forme de booléen. Une alternative est l'utilisation de

Fenêtre Scilab
```
-->exists('A')
 ans  =

    1.
```

avec la réponse qui vaut 0 ou 1 (faux vrai).

Scilab fournit des commandes utiles pour gérer les variables on peut citer

Fenêtre Scilab

```
-->who
your variables are...

 A          ans        startup    ierr       MODE_X     scicos_pal
 %helps     MSDOS      home       PWD        TMPDIR     plotlib
 percentlib            soundlib   xdesslib   utillib    tdcslib    siglib
 s2flib     roblib     optlib     metalib    elemlib    commlib    polylib
 autolib    armalib    alglib     intlib     mtlblib    SCI        %F
 %T         %z         %s         %nan       %inf       old
 newstacksize          $          %t         %f         %eps       %io
 %i         %e
 using        5288 elements  out of     1000000.
          and           47 variables out of        1791
```

La fonction **who** renvoie la liste des variables (on retrouve quelques connaissances "`A`", "`%i`", "`%e`" et "`%eps`"), des informations sur la place mémoire occupée et restante en nombre de mots (1 mot = 1 nombre double précision), et en dernière ligne le nombre de variables utilisées et disponibles (total).

Les commande **whos()**, **whos -name** 'début du(des) nom(s) de variable(s)', **whos -type** 'type de variable' renvoient des informations plus détaillées sur la taille des variables.

Fenêtre Scilab

```
-->whos -name %
Name                      Type            Size            Bytes
%F                        boolean         1 by 1          24
%T                        boolean         1 by 1          24
%z                        polynomial      1 by 1          56
%s                        polynomial      1 by 1          56
%nan                      constant        1 by 1          24
%inf                      constant        1 by 1          24
%t                        boolean         1 by 1          24
%f                        boolean         1 by 1          24
%eps                      constant        1 by 1          24
%io                       constant        1 by 2          32
%i                        constant        1 by 1          32

 -->whos -type boolean
Name                      Type            Size            Bytes

MSDOS                     boolean         1 by 1          24
%F                        boolean         1 by 1          24
%T                        boolean         1 by 1          24
%t                        boolean         1 by 1          24
%f                        boolean         1 by 1          24
```

Une fonction très utile sur les matrices, et donc sur les variables est la fonction **size**. Notez que **Scilab** peut faire des matrices de presque tout !

Fenêtre Scilab
```
-->size(A)
 ans  =

!   2.    2. !
```

**size** renvoie le nombre de lignes et colonnes.

**A faire :** `size(2)`. Alors ? Regarder également : `length(A)`, `typeof(A)`, `a=2*A`.

### 2.1.3 Suppression

Il est possible de supprimer une variable de l'environnement par la commande :

Fenêtre Scilab
```
-->clear A
```

ou

Fenêtre Scilab
```
-->clear
```

qui cette fois supprime toutes les variables non protégées.

Nous préciserons ultérieurement l'existence de variables locale et globale avec leurs utilisations et propriétés respectives.

## 2.2 Tableaux

On parlera de tableaux au lieu de vecteurs colonne ou ligne.

### 2.2.1 Génération

La génération de tableaux (ou vecteurs) peut se faire “manuellement” par une expression de la forme vue précédemment

Fenêtre Scilab
```
-->A=[1 2 3 4 5],B=[0 2 4 6 8 10],D=[]
```

avec des espaces (ou virgules) séparant les éléments pour une écriture ligne et de points virgules pour une écriture colonne. Il est possible d'avoir un tableau vide (D).

Une syntaxe plus adaptée à la création des tableaux A et B et la suivante

Fenêtre Scilab
```
-->A=1:5,B=0:2:10
 A  =

```

```
!   1.    2.    3.    4.    5. !
 B  =

!   0.    2.    4.    6.    8.    10. !
```

Autrement dit

**début : borne supérieure**

pour avoir un incrément automatique de 1 entre début et borne supérieure, ou alors

**début : incrément : borne supérieure**

pour spécifier l'incrément.

Attention cette dernière commande possède deux comportements à connaître , le premier est

Fenêtre Scilab
```
-->2:1
 ans  =

     []
```

où la réponse est le tableau vide. Le second

Fenêtre Scilab
```
-->1:1.1:4
 ans  =

!   1.    2.1    3.2 !
```

on voit ici que la dernière valeur est celle qui est inférieure à la borne indiquée. Cette dernière spécification peut engendrée des difficultés par exemple

Fenêtre Scilab
```
-->0:1/3:1
 ans  =

!   0.    0.3333333    0.6666667    1. !
```

tandis que

Fenêtre Scilab
```
-->0:1/3+2*%eps:1
 ans  =

!   0.    0.3333333    0.6666667 !
```

Pour palier à cette particularité il existe la commande **linspace**

Fenêtre Scilab
```
-->linspace(0,1,4)
 ans  =

!   0.     0.3333333    0.6666667    1. !
```

qui s'emploie sous la forme **linspace(début,fin,nombre de valeurs)**. Cette fois le tableau résultat contient forcément les valeurs limites.

Voir également **logspace(début,fin,nombre de valeurs)**[2].

### 2.2.2 Extraction et affectation

Bien sûr, avoir des tableaux sans la possibilité d'accéder aux valeurs ne serait pas du plus grand intérêt. On a vu l'usage des crochets [ et ] pour la création de tableaux, on utilisera pour l'extraction les parenthèses ( et ) exemples :

Fenêtre Scilab
```
-->t=1:5;
-->t(1),t(4)
 ans  =

    1.
 ans  =

    4.
```

L'accès aux éléments se fait donc simplement et naturellement, noter que les indices commencent à 1 (héritage du fortran).

Ces accès aux éléments du tableau permettent également d'affecter (tout ou) une partie du tableau :

Fenêtre Scilab
```
-->t(1)=5
 t  =

!   5.    2.    3.    4.    5. !
```

L'extraction et l'affectation jouent un rôle identique le tout étant de pointer sur le(s) élément(s) choisi(s).

Fenêtre Scilab
```
-->t(1)=t(5)
 t  =

!   5.    2.    3.    4.    5. !
```

[2]Voir le help correspondant pour les valeurs de début et de fin.

Nous avons déjà utilisé la fonction size, pour les tableaux la fonction **length** parait plus judicieuse, renvoyant la longueur totale de l'objet (même pour une matrice).

Fenêtre Scilab
```
-->length(t)
 ans  =

    5.
```

Néanmoins il existe une petite astuce pour atteindre le dernier élément d'un tableau, c'est l'utilisation dans l'expression de **$** :

Fenêtre Scilab
```
-->t($)
 ans  =

    5.
```

Nous avons ici les fonctions standards d'extraction que possèdent la plupart des langages de programmation, mais des langages tels que le fortran 90 ou Matlab[3] offrent la possibilité de directement pointer sur un sous-ensemble d'un tableau :

Fenêtre Scilab
```
-->t(2:5)
 ans  =

!   2.    3.    4.    5. !

-->t([3 4])
 ans  =

!   3.    4. !
```

ici on met entre parenthèses un ensemble d'indices. Ceci est valable à l'extraction comme à l'affectation.

Fenêtre Scilab
```
-->t(1:3)=3:-1:1
 t  =

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

A noter quelques particularités, la possibilité est offerte d'avoir redondance des indices avec deux comportements :

- à l'extraction

[3]Le C++ permet également de définir des extractions multiples sur des objets.

```
Fenêtre Scilab
-->t([2 2])
 ans  =

!   2.    2. !
```

- à l'affectation :

```
Fenêtre Scilab
-->t([1 1])=1:2
 t  =

!   2.    2.    1.    4.    5. !
```

avec au final la dernière valeur affectée (dans l'ordre des indices).

**A faire :**

- `t(1 :2 :$)`,
- `t($ :-1 :1)`
- `t( :)`

Regardons une autre particularité. Si on essaye d'extraire la valeur du 6ème élément de t alors

```
Fenêtre Scilab
-->t(6)
      !--error    21
invalid index
```

ce qui était prévisible, par contre

```
Fenêtre Scilab
-->t(6)=1
 t  =

!   2.    2.    1.    4.    5.    1. !
```

affecte dynamiquement une valeur supplémentaire à t. Maintenant faisons

```
Fenêtre Scilab
-->t(10)=1
 t  =

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

on constate que automatiquement est ré-alloué un tableau de taille suffisante complété de 0.

**Remarque :** L'utilisation intensive de cette (re)allocation dynamique peut engendrer des temps d'exécution prohibitifs.

Il est possible également de contracter l'écriture de certaines affectations

Fenêtre Scilab
```
-->t(1:6)=2
 t  =

!   2.    2.    2.    2.    2.    2.    0.    0.    0.    1. !
```

la partie entre parenthèses étant vue comme un ensemble d'indices pour lequel on affecte la valeur 2. Suivant ce principe on peut allouer des valeurs en dehors de la taille du tableau comme précédemment.

Par contre si la variable n'est pas pré-définie, l'interpréteur renvoie un tableau colonne.

Fenêtre Scilab
```
-->T(1:5)=1
 T  =

!   1. !
!   1. !
!   1. !
!   1. !
!   1. !
```

**A Faire :** `t=[], t($+3)=1`.

### 2.2.3 Concaténation

Précédemment nous venons de voir qu'il est possible d'affecter (ou d'extraire) la valeur d'un tableau par l'indice correspondant. Scilab reconnaît une autre forme syntaxique qui est la suivante :

Fenêtre Scilab
```
-->a=[1 2:5]
 a  =

!   1.    2.    3.    4.    5. !

-->b=[a  6:7]
 b  =

!   1.    2.    3.    4.    5.    6.    7. !
```

on peut "concaténer" deux ou plusieurs vecteurs en les écrivant entre crochets, cela peut donner des variantes du type

Fenêtre Scilab
```
-->a=[0 a 0]
 a  =

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

Cette forme d'écriture peut être utilisée lors de l'extraction

Fenêtre Scilab
```
-->[x,y,z]=(1,2,3)
 z  =

    3.
 y  =

    2.
 x  =

    1.
```

ou avec des tailles de vecteurs différentes

Fenêtre Scilab
```
-->[x,y,z]=(1:2,2:3,3:5)
 z  =

!   3.    4.    5. !
 y  =

!   2.    3. !
 x  =

!   1.    2. !
```

Cette écriture contractée peut être utilisée pour permuter des variables

Fenêtre Scilab
```
-->[x,y,z]=(z,x,y)
 z  =

!   2.    3. !
 y  =

!   1.    2. !
 x  =

!   3.    4.    5. !
```

### 2.2.4 Suppression d'éléments

La suppression d'un élément se fait à l'aide de [ ]

```
Fenêtre Scilab
-->a=1:5; a(3)=[]
 a  =

!   1.    2.    4.    5. !
```

comme pour l'affectation on peut faire) une suppression multiple

```
Fenêtre Scilab
-->a=1:5; a(1:2:5)=[]
 a  =

!   2.    4. !
```

Par contre la commande

```
Fenêtre Scilab
-->a(:)=[]
 a  =

     []
```

ne supprime pas la variable mais la conserve vide, pour faire disparaître la variable il faut utiliser la commande **clear**.

### 2.2.5 Primitives Scilab

Les fonctions usuelles s'appliquent sur les tableaux élément par élément.
Il existe quelques fonctions spécifiques :
- **sum()** : somme des éléments.
- **prod()** : produit des éléments.
- **mean()** : moyenne des éléments.
- **max()**, **min()** : la valeur maximale et minimale.
- **cumsum()**, **cumprod()** : renvoient un tableau avec pour le premier la somme cumulée des éléments et le second le produit cumulé.
- **sort()**, **gsort()**, **lex_sort()** : différentes procédures de tri.

### 2.2.6 Exercices

1. Faire `sum([])`.
2. Faire `prod([])`.
3. Écrire simplement $5!$ avec la fonction **prod**.
4. Initialiser le vecteur de taille 10 avec $v_i = \frac{1}{i}$.
5. Générer un vecteur aléatoire $v$ de taille 10 (`rand(1,10)`) et le normaliser pour la norme euclidienne ($\|v\| = \sqrt{\sum_i x_i^2}$).

6. Générer le tableau des puissances de 2 jusqu'à $2^8$ ;
7. Écrire un vecteur contenant les valeurs de $\sin(n2\pi/N)$, $n = 1..N$, pour $N = 20$.
8. Inverser l'ordre du vecteur précèdent.
9. Générer un vecteur de nombres allant de 1 jusqu'à 3 par pas de 0.2.
10. Extraire du précédent vecteur les rangs multiples de 5.

## 2.3 Matrices

### 2.3.1 Génération

Comme pour les tableaux, on peut définir "manuellement" les matrices. Mais il y a quelques primitives qui peuvent faciliter cette tâche.

- **zeros(n,p)** retourne la matrice nulle de n lignes et p colonnes.

Fenêtre Scilab
```
-->zeros(2,3)
 ans  =

!   0.    0.    0. !
!   0.    0.    0. !
```

- **ones(n,p)** comme précédemment mais la matrice est remplie de 1.

Fenêtre Scilab
```
-->ones(3,2)
 ans  =

!   1.    1. !
!   1.    1. !
!   1.    1. !
```

- **eye(n,p)** matrice identité de taille n $\times$ p ;
- **diag(v) ou diag(v,i)** Constitue à l'aide du vecteur ou tableau v la matrice possédant v comme diagonale principale. l'argument i spécifiant la diagonale concernée.

Fenêtre Scilab
```
-->diag(1:3)
 ans  =

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

-->diag(1:2,1)
 ans  =
```

```
!   0.    1.    0. !
!   0.    0.    2. !
!   0.    0.    0. !

-->diag(1:2,-1)
 ans  =

!   0.    0.    0. !
!   1.    0.    0. !
!   0.    2.    0. !
```

- **rand(n,p)** retourne une matrice de nombres aléatoires de loi uniforme sur $[0, 1]$.

Fenêtre Scilab

```
-->rand(2,2)
 ans  =

!   0.2113249    0.0002211 !
!   0.7560439    0.3303271 !
```

Pour obtenir une loi normale, ou autre, consulter le help correspondant.

- **matrix(v,n,p)** fonction de (re)formatage, parcours par colonne les éléments de v, qui sont mis colonne par colonne dans la matrice de sortie

Fenêtre Scilab

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

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

--> matrix(a,1,6)
 ans  =

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

--> matrix(a,3,2)
 ans  =

!   1.    5. !
!   4.    3. !
!   2.    6. !
```

L'argument -1 peut remplacer un des numéro de ligne ou colonne, par exemple la mise en colonne de la matrice

Fenêtre Scilab

```
--> matrix(a,-1,1)
```

```
 ans  =

!   1. !
!   4. !
!   2. !
!   5. !
!   3. !
!   6. !
```

Les primitives **ones()**, **zeros()** et **eye()** peuvent être utilisées en passant comme argument une matrice. Le résultat possède les mêmes dimensions que la matrice en argument.

```
Fenêtre Scilab
-->A=rand(2,2); ones(A)
 ans  =

!   1.    1. !
!   1.    1. !
```

### 2.3.2 Extraction et affectation

Comme pour les tableaux on peut extraire un élément par la syntaxe

```
Fenêtre Scilab
-->A=eye(3,3); A(1,1)
 ans  =

    1.
```

Mais on peut spécifier des ensembles d'indices :

```
Fenêtre Scilab
-->A([1 2],[2 3])
 ans  =

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

on extrait la sous matrice intersection des ligne 1 et 2 avec les colonnes 3 et 4. Cette syntaxe est identique à l'affectation

```
Fenêtre Scilab
-->A([1 2],[2 3])=2*ones(2,2)
 A  =

!   1.    2.    2. !
!   0.    2.    2. !
!   0.    0.    1. !
```

Comme pour les tableaux on peut utiliser la redondance des indices

Fenêtre Scilab
```
-->A([1 1 1],:)
 ans  =

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

Remarquer l'usage de : spécifiant tous les indices de colonne (ou ligne). On peut également se servir de $

Fenêtre Scilab
```
-->A(:,$)
 ans  =

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

Il est possible de considérer une matrice comme un vecteur, ce vecteur est composé des colonnes de la matrice mises bout à bout :

Fenêtre Scilab
```
-->A(3)
 ans  =

    0.

-->A(1:$)
 ans  =

!   1. !
!   0. !
!   0. !
!   2. !
!   2. !
!   0. !
!   2. !
!   2. !
!   1. !
```

**Remarque :** Cette représentation de la matrice sous forme de vecteurs colonnes correspond au mode de stockage des matrices (colonnes par colonnes).

### 2.3.3 Concaténation

Il est possible de tirer partie de la structure bloc d'une matrice pour sa définition

Fenêtre Scilab
```
-->A=[1:3; eye(2,2) [4;5] ]
 A  =

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

### 2.3.4 Suppression d'éléments

On ne peut pas, à proprement parler, supprimer un élément d'une matrice mais on peut supprimer une ou plusieurs rangée

Fenêtre Scilab
```
-->A=matrix(1:9,3,3); A(:,[2 3])=[]
 A  =

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

### 2.3.5 Primitives Scilab

Les fonctions **sum**, **cumsum**, **prod**, **cumprod**, **mean**, **max**, **min** possèdent trois syntaxes, une agissant sur la matrice entière et les deux autres respectivement sur les lignes (row) et les colonnes (colum)

Fenêtre Scilab
```
-->A=rand(3,3)
 A  =

!   0.2113249    0.3303271    0.8497452 !
!   0.7560439    0.6653811    0.6857310 !
!   0.0002211    0.6283918    0.8782165 !

-->max(A,'r')
 ans  =

!   0.7560439    0.6653811    0.8782165 !

-->max(A,'c')
 ans  =
```

```
!   0.8497452 !
!   0.7560439 !
!   0.8782165 !

-->max(A)
 ans  =

    0.8782165
```

On peut ajouter des fonctions d'extraction comme :
- **diag(A)**[4] : extrait la diagonale de la matrice A
- **tril(A)** et **triu(A)** : renvoyant respectivement une matrice composée de la partie inférieure (lower) et supérieure (upper) de A.

Comme **Scilab** est un environnement de manipulation matriciel, on retrouve la plupart des outils de calcul matriciel tels :
- **det()** Calcul du déterminant.
- **trace()** Somme des éléments diagonaux.
- **inv()** Inverse d'une matrice.
- **spec()** Calcul des vecteurs et valeurs propres.
- **rank()** Calcul du rang d'une matrice.
- **qr()** Décomposition QR d'une matrice.
- **lu()** Décomposition LU (élimination de Gauss) d'une matrice.
- **chol()** Décomposition de Cholesky d'une matrice.

et quelques autres fonctions...

### 2.3.6 Exercices

1. Générer une matrice de taille `10x10` diagonale de 2.
2. Inverser les lignes ou colonnes pour obtenir à partir de la matrice précédente une matrice antidiagonale.
3. Générer une matrice aléatoire de taille `3x4`
   - Permuter la dernière colonne avec la première.
   - Extraire le premier bloc `3x3` de la matrice.
   - Assembler la matrice de taille `6x4` fait de la superposition (l'une sur l'autre) de la matrice.
4. Assembler les matrices suivante

$$\begin{pmatrix} 1. & 1. & 1. & 2. \\ 1. & 1. & 1. & 3. \\ 1. & 1. & 1. & 4. \\ 1. & 2. & 3. & 4. \end{pmatrix} \begin{pmatrix} 1. & 4. & 7. \\ 2. & 5. & 8. \\ 3. & 6. & 9. \end{pmatrix} \begin{pmatrix} 2. & 1. & 2. \\ 1. & 1. & 1. \\ 2. & 1. & 2. \end{pmatrix} \begin{pmatrix} 1. & 1. & 1. & 1. \\ 2. & 2. & 2. & 2. \\ 3. & 3. & 2. & 3. \\ 4. & 4. & 4. & 4. \end{pmatrix}$$

[4]Cette commande a un comportement distinct quand une matrice ou un vecteur est passée en argument.

5. Générer 2 matrices aléatoires de taille `4x4`, former la matrice dont chaque élément et le sup des éléments correspondants des 2 premières matrices ($c_{ij} = \sup(a_{ij}, b_{ij})$).

## 2.4 Booléens

### 2.4.1 Opérateurs de comparaison

On a déjà vu dans la prise en main que le type booléen pouvait se composer en matrice avec les conjonctions & (et), | (ou) et ∼ (non) (attention à la compatibilité des dimensions !).

Mais la capacité matricielle des opérateurs de conjonction, offrant la possibilité de travailler avec des matrices booléennes s'étend aux opérateurs de comparaisons :

Fenêtre Scilab
```
-->A=1:5; A>3
 ans  =

! F F F T T !
```

Les opérateurs de comparaison sont >, >=, <, <=, == (égalité), ∼= ou <> (différent).

De plus il existe des version vectoriel des opérateurs de conjonction

Fenêtre Scilab
```
-->M=(1:5)>3
 M  =

! F F F T T !
-->or(M)
 ans  =

  T
-->and(M)
 ans  =

  F
```

Ces commandes ce déclinent également sous forme matricielle, avec le commutateur de ligne ou de colonne.

Fenêtre Scilab
```
-->M=rand(3,3)>0.4
 M  =

! F F T !
! T T T !
! F T T !

-->and(M,'r')
 ans  =
```

```
! F F T !

-->and(M,'c')
 ans  =

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

## 2.4.2 Génération

Comme vu dans la section précédente les opérateurs de comparaisons sont les principaux outils pour la construction de matrices de booléens. Néanmoins on peut rendre booléen certaines expression :

Fenêtre Scilab

```
-->a=round(rand(3,3))
 a  =

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

-->a|a
 ans  =

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

Cet exemple montre que les valeurs nulle d'une matrice sont interprétées comme la valeur booléenne fausse. Tout ce qui n'est pas nul est vrai. On peut par ce jeu créer des vecteurs ou matrices de booléens avec une syntaxe peu esthétique mais fonctionnelle :

Fenêtre Scilab

```
-->v=~ones(1,5)
 v  =

! F F F F F !

-->u=~zeros(1,7)
 u  =

! T T T T T T T !
```

```
-->w=~round(rand(1,7))
 w  =

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

Il est toujours possible d'utiliser des fonctions classiques comme les commandes **matrix()**, **diag()** ...

Fenêtre Scilab

```
-->d=~zeros(1,4)
 d  =

! T T T T !

-->B=diag(d)
 B  =

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

De manière plus anecdotique, il existe différentes passerelles entre le calcul booléen et le calcul réel. Il est possible de sommer un booléen avec un réel les valeurs 0 et 1, faisant office de valeurs fausses et vraies.

### 2.4.3 Compression

Scilab offre la possibilité d'extraire d'une matrice tout ou une partie à l'aide d'une matrice booléenne, spécifiant les éléments à retenir

Fenêtre Scilab

```
-->A=rand(2,2)
 A  =

!   0.2113249    0.0002211 !
!   0.7560439    0.3303271 !

-->T=[%t %f;%f %t]
 T  =

! T F !
! F T !

-->A(T)
 ans  =

```

```
!   0.2113249 !
!   0.3303271 !
```

Ceci peut donner lieu à des expressions de la forme

Fenêtre Scilab
```
-->A(A>0.5)
 ans  =

    0.7560439
```

### 2.4.4 Primitives Scilab

Il y a une primitive particulièrement utile, c'est la commande **find** qui s'applique sur des tableaux ou matrices booléennes et renvoie le(s) indices correspondants aux éléments vrais

Fenêtre Scilab
```
-->T=[%t %f %f %t]
 T  =

! T F F T !

-->find(T)
 ans  =

!   1.    4. !
```

La commande **find** peut également renvoyer les indices de lignes et colonnes de la recherche.

**A faire :**

- `A=rand(2,3)`
- `find(A>0.5)`
- `[i,j]=find(A>0.5)`

### 2.4.5 Exercices

1. Générer 2 matrices aléatoires de taille `5x5` (A et B). Trouver la matrice booléenne correspondant aux élément de A supérieurs à ceux de B (vrai si $a_{ij} > b_{ij}$).
2. Générer 1 matrice aléatoire de taille `5x5`, extraire les éléments compris entre 0.3 et 0.7.
3. Générer 1 vecteur aléatoire de taille `1x10`, soustraire 1 aux éléments plus grand que 0.5.
4. Générer 1 matrice aléatoire de taille `5x5` de nombre entier compris entre 0 et 10. Trouver les éléments égaux à 0.

## 2.5 Polynômes et Fractions rationnelles

Scilab permet la manipulation des polynômes et fractions rationnelles définissant pour cela deux types : **polynomial** et **rational**.

### 2.5.1 Génération

Scilab possède une variable pré-définie : **%s**. Cette dernière représente le monôme de degré 1 et permet la création de polynômes ou de fractions rationnelles par une syntaxe de la forme

Fenêtre Scilab

```
-->p=1+%s+2*%s^2
 p  =

              2
   1 + s + 2s

-->q=p/(1 + %s)
 q  =

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

**A faire :** `typeof(p)`, `typeof(q)`, `q('num')`, `q('den')` avec `p` et `q` définis ci-dessus.

La possibilité est offerte avec la commande **poly** de directement définir un polynôme par ses racines (par défaut) ou bien par ses coefficients

Fenêtre Scilab

```
-->r=poly([0 1 2],'s')
 r  =

           2   3
   2s - 3s + s

-->r2=poly([0 1 2],'s','coeff')
 r2  =

          2
   s + 2s
```

Les commandes **roots()** et **coeff()** permettent les opérations inverses (ou effectue le calcul des racines pour la première).

Les commandes algébriques usuelles : somme, multiplication, division et puissance sont valides.

**Remarque1 :** le caractère 's' (variable symbolique) a été utilisé dans la définition du polynôme r, l'utilisateur a le choix de sa variable polynomiale, par contre les compositions (somme ...) nécessitent l'emploi de polynômes possédant la même variable

Fenêtre Scilab

```
-->poly([0 1 2],'x') + poly([1 1],'s')
                                    !--error     4
undefined variable : %p_a_p
```

La commande **varn()** peut résoudre se problème.

**Remarque2 :** la commande **poly** peut prendre une matrice carrée comme argument, le résultat est dans ce cas le polynôme caractéristique de la matrice

Fenêtre Scilab

```
-->poly([2 1; 1 2],'x')
 ans  =

              2
   3 - 4x + x
```

si la matrice n'est pas carrée alors les éléments sont lus colonne par colonne et c'est donc le polynôme défini par ces racines qui est retourné.

### 2.5.2 Évaluation

Pour évaluer un polynôme (et fraction), la commande **horner(polynôme,valeurs)** intervient

Fenêtre Scilab

```
-->p=poly([1 1 2],'x'); horner(p,[0 0.5])
 ans  =

! - 2.  - 0.375 !

-->horner(p,[0 1;1 0.5])
 ans  =

! - 2.    0.     !
!   0.  - 0.375 !
```

### 2.5.3 Primitives Scilab

Scilab possède quelques fonctions dédiées au calcul des polynômes et fractions.

- **degree()** : renvoie le degré du polynôme (pas de la fraction !).
- **derivat()** : procède à la dérivation du polynôme ou fraction.
- **coeff()** : retourne le tableau des coefficients du polynôme classés par ordre croissant.
- **lcm( , ), gcm( , )** : plus petit et plus grand multiple commun.
- **pdiv(), ldiv()** : division euclidienne et division par ordre croissant.
- **roots()** : extrait ou calcule les racines du polynôme.
- **simp()** : simplification de fraction.
- **factors()** : factorisation en éléments simples.

### 2.5.4 Exercices

1. Écrire le polynôme $p(x) = 1 + x + 3x^2 - x^3$.
2. Écrire le polynôme $q(x) = (x - 1)(x - 2)(x - 3)$, Calculer les racines de $q$.
3. Même question que précédemment mais avec $q(x) = \prod_{i=1}^{n}(x - i)$, pour $n = 15, 20, 25$
4. Écrire la fraction $f(x) = \frac{x(1-2x)}{(x+1)(x-1)}$.
   - Calculer *df* la dérivée de $f$.
   - Évaluer *df* aux points $x = 0, 2, 3$ et $4$.
   - Trouver les pôles de *df*.

## 2.6 Chaîne de caractère(s)

### 2.6.1 Génération

Les chaînes de caractères constituent un type Scilab , et à ce titre, elles admettent des règles d'usage et de comportement.

Une chaîne de caractères est une expression délimitée par simple apostrophe (') ou double apostrophe (").

Fenêtre Scilab
```
-->a='coucou' , b="hello"
 a  =

 coucou
 b  =

 hello
```

pour faire apparaître la simple ou double apostrophe dans l'expression, il faut doubler l'apostrophe

Fenêtre Scilab
```
-->c='voici une apostrophe : ''  une double apostrophe "".'
 c  =

 voici une apostrophe : '  une double apostrophe ".
```

Il est possible de transformer toute expression numérique en chaîne de caractères avec la commande **string()**.

Fenêtre Scilab
```
-->string(tan(0.5))
 ans  =

 0.5463025

-->typeof(ans)
 ans  =

 string
```

**Remarque :** Attention les caractères accentués de notre chère langue ne donnent pas toujours le rendu escompté[5]

Fenêtre Scilab
```
-->c='é'
 c  =

 !
```

La chaîne de caractères, comme la plupart des types Scilab, admet une syntaxe matricielle.

Fenêtre Scilab
```
-->A=["une" "matrice" ; "de" "caractères"]
 A  =

!une  matrice      !
!                  !
!de   caracteres  !
```

### 2.6.2 Concaténation

La concaténation se fait à l'aide du signe +

Fenêtre Scilab
```
-->a="tan(0.5)="+string(tan(0.5))
 a  =

 tan(0.5)=0.5463025
```

Dans le cas d'une matrice on peut utiliser la commande **strcat()** qui permet de mettre bout à bout les éléments de la matrice (parcours par colonnes)

[5]Ceci est maintenant faut depuis le version 2.7 de **Scilab**

Fenêtre Scilab
```
-->A=["une" "de";"matrice" "caracteres"];

-->strcat(A)
 ans  =

 unematricedecaracteres

-->strcat(A," ")
 ans  =

 une matrice de caracteres
```

Dans ce dernier cas la concaténation est effectuée avec un espace intercalé entre chaque terme.

la concaténation admet des extensions naturelles entre une matrice (de caractères) et un scalaire (également chaîne de caractères)

Fenêtre Scilab
```
-->A=['a';'b';'c']
 A  =

!a  !
!   !
!b  !
!   !
!c  !

-->'(' + A + ')'
 ans  =

!(a)  !
!     !
!(b)  !
!     !
!(c)  !
```

### 2.6.3 Extraction

La commande **part()** permet d'extraire tout ou une partie d'une chaîne de caractères

Fenêtre Scilab
```
-->a="hello"
 a  =

 hello
```

```
-->part(a,1)
 ans  =

 h

-->part(a,[1 3 5])
 ans  =

 hlo
```

le tableau passé en paramètre correspondant aux positions des caractères à extraire. Cette commande est souvent utilisée avec la commande **length** qui permet d'avoir la longueur de chaîne (attention au cas matriciel).

la commande **strindex()** retourne la position du caractère ou chaîne spécifié en argument.

Fenêtre Scilab

```
-->strindex(a,'l')
 ans  =

!   3.    4. !

-->strindex(a,'ll')
 ans  =

    3.
```

### 2.6.4 Primitives Scilab

Scilab possède un jeu restreint de fonctions dédiées à la manipulation de chaînes de caractères.

- **convstr()** : change la "case" de la chaîne de caractère ("u" ou "l").
- **grep()** : recherche une chaîne de caractère dans une autre ou tableau de chaînes de caractères.
- **length()** : la fonction length appliquée à une chaîne de caractères n'a pas le même comportement que sur les autres types. Dans ce cas, elle renvoie une matrice contenant les tailles des chaînes de l'argument.
- **str2code()** et **code2str()** : le première fonction renvoie le tableau de code Scilab des caractères, et la seconde fait l'opération inverse.
- **strsub()** : substitue une chaîne de caractères dans une autre.
- **stripblanks()** : supprime les espaces dans l'expression.

A noter la commande **evstr** qui évalue une chaîne de caractère :

Fenêtre Scilab
```
-->a=1; b='2*a' ; evstr(b)
 ans  =

    2.
```

ou encore la commande **execstr** qui demande à l'interpréteur l'exécution **Scilab** de cette chaîne :

Fenêtre Scilab
```
-->execstr('a=%t')

-->a
 a  =

  T
```

Ces dernières commandes sont relativement anodines, néanmoins, comme nous somme dans un environnement interprété, cela peut donner un caractère "dynamique" au programme, permettant même de s'auto-générer !

### 2.6.5 Exercices

1. Former la chaîne de caractère `"1,2,3,4,5"`.
2. Trouver les positions de `'o'` dans `'bonjour'`, remplacer les occurrences de `'o'` par `'-'`.
3. Former la chaîne `'pi=3.1415927'` en utilisant la variable prédéfinie `%pi`.

## 2.7 Liste

Scilab offre la possibilité de construire des types "list". Ce type de variable permet de gérer des structures de données complexes. Il est possible ainsi dans une variable de type "list" de faire cohabiter tous types de variables, même d'autres listes, sans condition de compatibilité.

### 2.7.1 Génération

Pour obtenir une liste il suffit d'utiliser le mot réservé **list** sous la forme

Fenêtre Scilab
```
-->a=list(1,1:2,rand(2,3),"toto")
 a  =
       a(1)

    1.
```

```
      a(2)

!   1.    2. !

      a(3)

!   0.2113249    0.0002211    0.6653811 !
!   0.7560439    0.3303271    0.6283918 !

      a(4)

 toto
```

on obtient une liste indicée contenant nos éléments.

Il est possible de créer une liste vide

```
Fenêtre Scilab
-->b=list()
 b  =

     ()
```

### 2.7.2 Affectation-Extraction

Le type list se comporte comme un tableau mono-dimensionnel, chaque élément étant référencé par son numéro de liste[6]. D'où pour obtenir un des éléments on a la syntaxe d'extraction

```
Fenêtre Scilab
-->a=list(1,1:2,"toto");
-->a(2)
 ans  =

!   1.    2. !
```

L'affectation d'un élément se fait de la même manière

```
Fenêtre Scilab
-->a(1)=5:10
 a  =
      a(1)

!   5.    6.    7.    8.    9.    10. !

      a(2)
```

[6] Comme pour les tableaux le premier élément de liste commence à 1 et pas 0

```
!   1.    2. !

      a(3)

 toto
```

On a vu que pour les tableaux on pouvait affecter (ou extraire) un élément ou une partie (ensemble d'indices). Pour le type list les choses ne sont plus tout à fait identiques. En effet si on affecte une partie de liste, alors la réponse est immédiate

```
Fenêtre Scilab
-->a(1:2)=[2,2:3]
               !--error    43
not implemented in scilab....
```

Par contre à l'extraction nous avons le comportement suivant

```
Fenêtre Scilab
-->a(1:2)
       !--error    41
incompatible LHS
```

mais

```
Fenêtre Scilab
-->[s,t]=a(1:2)
 t  =

!   1.    2. !
 s  =

    1.
```

il est donc permis de "multi-extraire" les éléments d'une liste avec l'utilisation des crochets.

### 2.7.3 Suppression d'éléments

Il existe une liste particulière c'est la liste **null()**. Pour supprimer un élément de liste il suffit de faire

```
Fenêtre Scilab
-->a=list(1,1:2,"toto");
-->a(2)=null()
 a  =
      a(1)

    1.
```

```
      a(2)

 toto
```

pour voir le deuxième élément de `a` disparaître. Noter le ré-ordonnancement automatique.

On ne peut toujours pas supprimer un ensemble d'indices

```
Fenêtre Scilab
-->a(1:2)=null()
              !--error    43
not implemented in scilab....
```

Mais attention le comportement de **null()** peut être radical

```
Fenêtre Scilab
-->a=list(1,1:2,"toto");
-->a=null()
-->a
  !--error     4
undefined variable : a
```

**Remarque :** l'élément **list()**, liste vide est une liste, mais pas **null()**.

On peut accéder aux champs d'un élément de la liste par un mode d'extraction hiérarchique. Dans l'exemple suivant le second élément de la liste est un tableau, voici la syntaxe pour directement obtenir la première valeur du tableau :

```
Fenêtre Scilab
-->a=list(1,1:2,"toto")    //definition de a
-->a(2)
 ans  =

!   1.    2. !

-->a(2)(1)
 ans  =

    1.
```

### 2.7.4 Ajout d'éléments

Le type list se comporte comme une liste séquentielle à multiples entrées (en début, fin ou à indice).

- **Ajout en fin de liste :**

Fenêtre Scilab
```
-->a=list(1:2);
-->a($+1)=2:3
 a  =
       a(1)

!   1.    2. !

       a(2)

!   2.    3. !
```

– **Ajout en début de liste :**

Fenêtre Scilab
```
-->a(0)="hello"
 a  =
       a(1)

 hello

       a(2)

!   1.    2. !

       a(3)

!   2.    3. !
```

Dans ce cas, les indices sont automatiquement décalés pour obtenir le premier élément à l'indice 1.

– **Ajout à l'indice :** Ceci est standard, de plus comme pour le cas de tableaux si on affecte un élément hors indice de liste, Scilab complète automatiquement jusqu'à l'indice de l'élément affecté

Fenêtre Scilab
```
-->a=list(1:2);
-->a(4)=2:3
 a  =
       a(1)

!   1.    2. !

       a(2)

    Undefined

       a(3)
```

```
   Undefined

      a(4)

!   2.    3. !
```

**Faire :** `a=list()`, `a($)=2 :3` , `a($+1)=2 :3`.

### 2.7.5 Liste de listes

Une liste peut contenir elle même une autre liste

Fenêtre Scilab

```
-->a=list(1,list("hello",1:2))
 a  =
      a(1)

   1.

      a(2)


       a(2)(1)

 hello

       a(2)(2)

!   1.    2. !
```

Dans ce cas on peut extraire ou affecter avec les deux indices de liste

Fenêtre Scilab

```
-->a(2)(2)=null()
 a  =
      a(1)

   1.

      a(2)


       a(2)(1)

 hello
```

On peut faire des listes de listes de listes ...

```
Fenêtre Scilab
-->a=list();
-->for i=1:5, a=list(a); ,end
-->a
 a  =
       a(1)


        a(1)(1)


         a(1)(1)(1)


          a(1)(1)(1)(1)


           a(1)(1)(1)(1)(1)

     ()
```

## 2.8 Liste typée

La liste typée (mot clef : **mlist**[7]) est un “objet” Scilab très intéressant. Elle possède :

- la flexibilité d'une liste, permettant de faire cohabiter différents genres (booléens, tableaux, listes ...).
- une syntaxe d'extraction et d'affectation propre.
- la possibilité de surcharger les opérateurs algébriques.

Cet ensemble de fonctionnalités permet d'étendre la structure algébrique à d'autres formes de représentation. On pourra consulter le help de `rational` pour voir une construction de fraction rationnelle (une fraction rationnelle est composée de deux polynômes, un “num”-érateur et un “den”-ominateur).

### 2.8.1 Déclaration

On parlera ici de déclaration de listes typées. En effet cette structure nécessite un peu plus d'attention lors de sa création. Il faut en spécifier les différents champs dès sa création, on ne peut en rajouter dynamiquement.

La syntaxe de déclaration est la suivante :

[7]Il existe un autre type **tlist** qui possède de prime abord les mêmes caractéristiques mais semble être plus pauvre et plus imparfait dans la gestion de surcharge d'opérateur.

*variable_type*=([*nom_type* ;*nom_champ1* ;*nom_champ2* ;...],*champ1*,*champ2* ,...)

- *nom_type* : chaîne de caractères définissant le nom du type (ex : `rational`).
- *nom_champ1*,*nom_champ2* : chaînes de caractères nommant les champs.
- *champ1*,*champ2* : les différents champs pouvant être de tout type Scilab (constante, matrice, booléen, list ou autre mlist). La présence de *champ1*,*champ2* n'est pas obligatoire.

**Exemple**

Fenêtre Scilab

```
-->fraction=mlist(['rational';'num';'den'],%s,1+%s)
 fraction  =


       fraction(1)

!rational  !
!          !
!num       !
!          !
!den       !

       fraction(2)

    s

       fraction(3)

    1 + s
```

On voit sur l'exécution de cet exemple qu'une liste typée est une liste, qui contient en premier élément un vecteur colonne de chaînes de caractères qui renseigne de la structure de la variable

**A faire :** Tapez `type(fraction)`,`typeof(fraction)`, noter le rôle joué par la fonction `typeof`.

## 2.8.2 Extraction et affectation

Il existe trois modes d'accès aux éléments. Le premier qui sera totalement oublié par la suite est l'accès par élément de list, ce qui peut donner sur l'exemple précédent :

Fenêtre Scilab

```
-->fraction(2)
 ans  =
```

```
    s

--> fraction(3)=1-%s


       fraction(1)

!rational  !
!          !
!num       !
!          !
!den       !

       fraction(2)

    s

       fraction(3)

    1 - s
```

Mais ceci est à proscrire, car l'ordre d'attribution dans la liste typée n'est pas en fonction de l'ordre des noms des différents champs, c'est l'ordre "d'arrivée" qui prédomine.

Les deux autres façons se font par le nom des champs et sont strictement équivalentes :

Fenêtre Scilab

```
 -->fraction.num
 ans  =

    s

-->fraction.den=1-%s
 fraction  =


       fraction(1)

!rational  !
!          !
!num       !
!          !
!den       !

       fraction(2)
```

```
    s

       fraction(3)

    1 - s
```

ou alors

Fenêtre Scilab

```
-->fraction('num')
 ans  =

    s

-->fraction(''den'')=1-%s
 fraction  =


       fraction(1)

!rational  !
!          !
!num       !
!          !
!den       !

       fraction(2)

    s

       fraction(3)

    1 - s
```

### 2.8.3 Surcharge d'opérateur

La surcharge d'opérateur (algébrique) peut permettre la construction "d'objets" pour lesquels on définit leurs comportements avec les opérateurs binaires de sommation, multiplication ...

L'opérateur unaire qui peut apporter un peu de confort est "l'opérateur" d'affichage. Le mécanisme d'affichage pour les listes typées est le suivant : l'interpréteur identifie s'il possède parmi ses variables la fonction *nom_type*_p, si celle-ci n'existe pas c'est la fonction d'affichage de list qui est usitée.

**Exemple** Pour notre exemple de fraction rationnelle on peut définir (en ligne dans Scilab 2.6) :

Fenêtre Scilab
```
function %rational_p(fr)
    disp(fr.num);
    disp('  --------');
    disp(fr.den);
endfunction
```

pour obtenir le résultat suivant :

Fenêtre Scilab
```
-->fraction
 fraction  =


    s

   --------

    1 - s
```

Remarquez que l'on utilise de manière imbriquée les différentes fonctions d'affichage des différents types (ici c'est la fonction du type `polynome`).

Si on cherche à sommer notre "rational" à un scalaire l'essai infructueux nous donne :

Fenêtre Scilab
```
-->1+fraction
    !--error     4
undefined variable : %s_a_rational
```

ici l'interpréteur de Scilab cherche la fonction `%s_a_rational()` correspondant à l'opérateur binaire de sommation ("_a_") entre un scalaire à gauche ("s") et un type `rational` à droite. Il paraît légitime d'informer Scilab sur nos intentions de sommation en définissant la fonction suivante :

Fenêtre Scilab
```
function out=%s_a_rational(s,f)
    out=f;
    out.num=out.num + s*out.den;
endfunction
```

Ainsi la somme peut se passer correctement.

$\boxed{\textbf{A faire :}}$ Après avoir fait les étapes proposées en exemple, faire `fraction +1`.

Le nom des fonctions de surcharge pour les opérateurs binaires suit la règle :

*%type1 _symbol_opération_ type2*

| symbole | type |
|---|---|
| b | booléen |
| c | caractère |
| p | polynôme |
| r | fraction rationnelle |
| m | fonction |
| s | constant |
| hm | hyper matrice |
| l | liste |
| *nom_type* | liste typée |
| sp | matrice "sparse" |
| spb | matrice "sparse" booléenne |

FIG. 2.1 – Symboles associés aux types prédéfinis

- *type1* et *type2* peuvent être des nouveaux types (listes typées) ou correspondre aux valeurs pré-définies suivantes :

- *symbole_opération*

| symbole | opérateur | opération |
|---|---|---|
| a | + | addition (non commutative) |
| s | - | soustraction (non commutative) |
| m | * | multiplication |
| r | / | division à droite |
| l | \ | division à gauche |
| p | ^ | exponentiation |
| t | ' | transposition |

FIG. 2.2 – Symboles de surcharge d'opérateur

# Chapitre 3

# Programmation

Au delà du fait d'avoir un interpréteur de commande en ligne, **Scilab** offre des possibilités étendues de programmation. De par son mode interactif, **Scilab** devient un langage de programmation interprété (et non pas compilé).

Ce côté interprété peu engendrer des temps d'exécution beaucoup plus long qu'un langage classique, car il faut ajouter au temps d'exécution, le temps d'interprétation des commandes. Mais un peu d'expérience permet de vectoriser[1] un grand nombre d'opérations et obtenir des temps raisonnables.

Ce désagrément est contrebalancé par deux avantages non négligeables. Le premier est lié à la syntaxe de **Scilab** qui permet une écriture efficace, pour des manipulations de tableaux et matrices, de tri et autres commandes intégrées. Cela permet, comparé à d'autre langages, d'économiser et de rendre plus concis ses programmes. Le second est dut au fait que nous sommes en permanence dans un espace de travail en mode interactif. Ce style "debuggage", donne accès à toutes les variables, permettant d'en modifier le contenu, tracer des graphiques... Ceci est l'avantage du mode interprété.

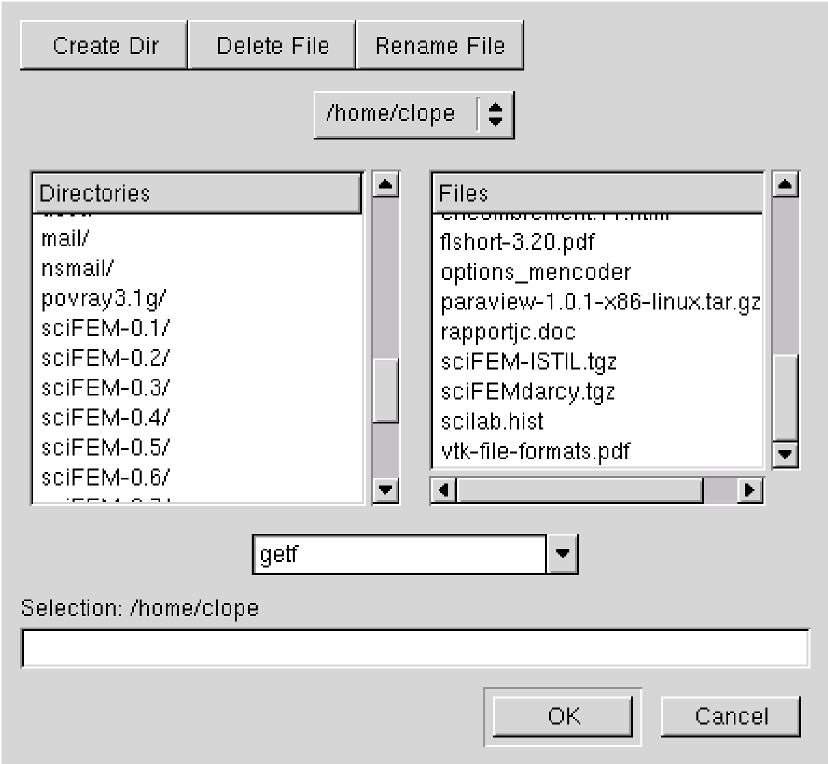
## 3.1 Script d'exécution

Un script est un fichier texte (Ascii) contenant une suite d'instructions **Scilab**, le mot programme est oublié pour laisser la place à des termes tels script principal, script d'exécution... Ce fichier est composé avec votre éditeur favori externe à **Scilab**, ce sera par exemple Emacs, Xemacs, Nedit ou ConText qui possèdent un mode d'édition spécifique à **Scilab**. Le choix d'un éditeur n'est pas fondamental, par contre il faudra faire attention de toujours finir le fichier avec un retour chariot sous peine que la dernière ligne ne soit pas interprétée.

[1] La vectorisation de calcul consiste à aligner des calculs arithmétiques semblables (exemple : multiplication terme à terme de deux tableaux), ceci, pour tirer partie de la structure de la machine (buffer, mémoire cache, temps d'accès mémoire ...). Dans notre cas, le mot vectorisation est légèrement détourné de cette définition. Il est plus proche au sens étymologique car d'une manière générale, l'utilisation ou l'écriture sous forme de vecteurs (ou partie de vecteur) soulage la charge de l'interpréteur et fait appel à des bibliothèques optimisées (BLAS). Par extension abusive, sera appelé vectorisation, tout processus permettant d'obtenir un coût d'interprétation et d'exécution le plus réduit possible.

Dans la suite on prendra pour convention de nom de fichier, l'extension ".sce" pour les scripts[2].

Une fois nos instructions enregistrées (dans test.sce par exemple), il nous faut l'exécuter. Pour cela on peut aller dans le menu de la fenêtre principale à File puis File operations pour voir apparaître




Ce qui permet de sélectionner le script de notre choix et de l'exécuter en cliquant sur le bouton Exec.

Cette méthode peut être délaissée au profit d'une commande en ligne de la forme

Fenêtre Scilab
```
-->exec test.sce
```

ou

Fenêtre Scilab
```
-->exec('test.sce')
```

Mais attention pour que cela se passe correctement il faut que le fichier soit dans le répertoire courant de Scilab, celui pointé par la variable **pwd**

Fenêtre Scilab
```
-->pwd
 ans  =

 /home/clopeau
```

[2]Seront utilisés deux types d'extensions : les ".sce" et les ".sci". Ce dernier sera réservé pour les fichiers de fonctions

Si notre fichier n'est pas localisé à cet endroit (commande **dir()** pour voir les fichiers), on peut changer de répertoire avec la commande **chdir()**

Fenêtre Scilab
```
-->chdir('Scilab')
 ans  =

    0.

-->pwd
 ans  =

 /home/clopeau/Scilab
```

Autrement, en dernier recours, il est possible de spécifier le nom complet avec répertoire[3]du script à exécuter.

## 3.2 Éléments de programmation

En premier lieu, il faut savoir qu'il est possible (voir recommandé) d'insérer des commentaires dans un script. Les commentaires sont délimités sur la gauche par le double slash et vont jusqu'à la fin de ligne.

script Scilab
```
// Ceci est une ligne complete de commentaire

A=[2 3; 1 2] //on peut mettre des commentaires apres des instructions
```

Si une instruction ne tient pas sur une ligne on peut utiliser l'opérateur ... pour l'écrire sur plusieurs lignes (cette remarque est valable pour les commandes en ligne), exemple :

script Scilab
```
a=sin(x+2*%pi)*tan(2*x)/...
  (1+x^2+log(1+x))^(asin(%pi/(1+x)))
```

Enfin une dernière règle est que chaque instruction se trouve, soit sur une ligne distincte, soit sur la même ligne que d'autres et sera séparée par une virgule ou un point virgule[4].

script Scilab
```
a=1; b=2 , c=3
```

Les instructions propres à la programmation ne sont pas très nombreuses. Elles ont toutes la propriété d'avoir une structure bloc, c'est à dire un mot clef d'ouverture (**if**, **for** ...) et un "**end**" de fermeture. Elle possèdent également la caractéristique d'être exécutable dans l'espace de travail, l'affichage des lignes devenant plus serré dans l'attente de l'instruction "**end**" pour évaluer le bloc. Nous allons donc expliciter ces instructions.

---

[3]De manière pratique, il est plus efficace de raisonner et de nommer les appels en relatif, du style : `../autr_repertoire`. Cela permet de changer de machines (répertoire), car une fois le répertoire de travail spécifié, il n'est plus besoin se soucier des noms inscrits dans les différents scripts.

[4]la virgule et le point virgule jouent le même rôle séparateur, le point virgule "bloque" l'affichage à l'exécution.

### 3.2.1 if ... elseif ... else ... end

L'instruction conditionnelle est gérée par les mots clefs : **if**, **elseif**, **else** et finalement **end**. **if** et **end** étant obligatoire ainsi que l'ordre (on peut répéter **elseif** autant que nécessaire).

script Scilab

```
if test1        // test1 est un booléen (scalaire)
   instruction1
elseif test2
   instruction2 // on peut sous cette forme mettre plusieurs lignes
else
   instruction3
end
```

Noter la structure bloc qui permet d'insérer une ou plusieurs ligne entre les mots clefs. On peut utiliser tout ou partie de cette structure

script Scilab

```
if test1
  instruction1
else
  instruction2
end
```

Il existe quelques variantes d'écriture de cette instruction conditionnelle notamment, une écriture ligne sous la forme

Fenêtre Scilab

```
-->x=1;
-->if x==2, x=1, elseif x==3, x=1, else x=2, end
 x  =

    2.
```

Noter l'usage des virgules après chaque booléen et avant chaque mot clef **elseif**, **else** et **end**, le point virgule pourrait jouer un rôle tout à fait identique (affichage en moins).

### 3.2.2 select ... case ... else ...end

Il existe une deuxième forme de choix conditionnel

script Scilab

```
select atester      // atester contient un expression
  case  cas1        // si atester vaut cas1 alors
     instruction1
  case  cas2        // si atester vaut cas2 alors
     instruction2
  else              // autrement
     instruction3
end
```

Cette structure permet d'écrire dans certain cas des arbres de décision plus agréables comme dans l'exemple ci-dessous

script Scilab
```
a=round(rand(1,2));
select a
  case [0 0]
    b=1;
  case [0 1]
    b=2;
  else
    b=0
end
```

Il est possible comme précédemment d'avoir une écriture sur une seule ligne de la forme

Fenêtre Scilab
```
-->a=round(rand(1,2));
-->select a, case [0 0],b=1,case [0 1],b=2,else,b=0 ,end
```

la règle étant de toujours respecter l'articulation de la syntaxe par l'emploie de la virgule (ou point virgule).

### 3.2.3 for ... end

la boucle **for** pointe sur un ensemble d'indices

script Scilab
```
for i=1:50
  // i vaut de 1 a 50
  bloc instructions
end
```

mais peut également se mettre sous la forme

script Scilab
```
for i=[2 7 1 4]
  bloc instructions
end
```

la variable i prenant tour à tour les valeurs 2,7, 1 et 4.

**Remarque :** La variable i est une variable locale à la boucle!

Il est possible de faire une boucle sur une matrice, dans ce cas le compteur est affecté à chaque colonne

script Scilab
```
for i=[2 3;1 2]
   i vaut [2;1] puis [3;2]
end
```

Comme pour le choix conditionnel, il est possible d'écrire la commande sur une seule ligne

Fenêtre Scilab
```
-->for i=1:5, disp(i); end
```

Enfin une autre possibilité d'utilisation de la boucle **for** est de boucler sur une liste d'éléments.

L'instruction **break** permet de sortir à tout moment de la boucle.

### 3.2.4 while ... end

L'instruction **while** est d'un usage simple sous la forme

script Scilab
```
while test  //test est un booleen scalaire
  bloc instructions
end
```

Le "bloc instructions" est exécuté tant que `test` est vrai.
L'instruction **break** permet de sortir à tout moment de la boucle.

Pour les amoureux de lignes peu espacées et aérées, il est toujours possible de faire contenir une boucle conditionnelle sur une seule ligne.

Fenêtre Scilab
```
-->while a<10, a=a+1; end
```

## 3.3 Fonctions

Une façon usuelle de définir des fonctions est de mettre celles-ci dans un fichier à extension ".sci". Le nom de fichier est sans importance[5], et plusieurs fonctions peuvent être dans le même fichier.

Néanmoins il est possible de définir "inline" une fonction avec la commande **deff**. Depuis la version 2.6 de **Scilab**, on peut directement dans un script écrire une fonction avec la même syntaxe comme si celle-ci se trouvait dans un fichier séparé. Elle est "chargée" automatiquement.

### 3.3.1 Syntaxe

la fonction doit commencer par le mot réservé **function** et finir par **endfunction** sous la forme

script Scilab
```
function [out1,out2,...,outN]=nomfonction(in1,in2,...,inP)

   // out1,out2,...,outN sont les variables de sortie
```

[5] Contrairement à Matlab.

```
   // in1,in2,...,inP variables d entree

   instructions
endfunction
```

Il faut veiller à ne pas "écraser" une fonction **Scilab** existante, dans ce cas doit apparaître un message signalant une telle éventualité.

Fenêtre Scilab
```
-->function out=norm(in)
-->  out=sqrt(sum(in.^2,'r'));
-->endfunction
Warning :redefining function: norm
```

Dans ce dernier cas la fonction est définie directement dans l'espace de travail, elle est automatiquement "chargée".

### 3.3.2 Charger une fonction

Pour que la fonction soit reconnue et exécutée par **Scilab** il faut la "charger". En effet les fonctions sous **Scilab** ont un statut de variables, et donc nécessitent d'être initialisées. Le fait qu'une fonction soit une variable permet, de manière naturelle de la passer en paramètre.

Pour cela comme pour l'exécution d'un script, par le menu File puis File opérations, on peut sélectionner le fichier (.sci) et cliquer sur getf.

On peut faire ceci en ligne par la commande **getf**

Fenêtre Scilab
```
-->getf NomFichier.sci
```

ou

Fenêtre Scilab
```
-->getf(''/nom.complet.du.repertoire/NomFichier.sci')
```

et les fonctions[6] de NomFichier.sci deviennent accessibles.

**Remarque :** Attention quand on modifie une fonction il faut la "recharger" (doit apparaître : Warning :redefining function ).

### 3.3.3 Appel d'une fonction

Pour exécuter une fonction il suffit de l'appeler en passant les arguments nécessaires

Fenêtre Scilab
```
-->mafonction(rand(2,2))
```

Mais dans le cas où ma fonction est définie comme suit

[6]Un fichier pouvant contenir plusieurs fonctions.

script Scilab
```
function [a,b]=mafonction(A)
      .....
```

l'appel précédent ne renvoie que la valeur de a.

Pour obtenir les deux valeurs escomptées il faut faire un appel sous la forme

Fenêtre Scilab
```
-->[p,q]=mafonction(rand(2,2))
 q  =

!   0.8782165    0.5608486 !
 p  =

!   1.           0. !
```

Ceci est déjà le cas pour un certain nombre de fonctions **Scilab**par exemple

Fenêtre Scilab
```
-->[m,k]=max(rand(1:10))
 k  =

    9.
 m  =

    0.8782165
```

où la variable `k` va contenir le rang du terme maximal.

**Scilab** offre la possibilité lors d'un appel de fonction de contrôler le nombre de paramètres en entrée et sortie. Il s'agit de la commande **argn**. Voici un exemple d'une fonction, qui teste le nombre d'argument passé en paramètre et calcul le déterminant de l'argument et sa trace dans le cas où deux variables de sortie sont spécifiées.

script Scilab
```
function [a,b]=mafonction(A)
  [lhs,rhs]=argn()
                        // lhs : left hand side
                        // rhs : right hand side
  if rhs==0
     disp('Impossible de faire quelque chose')
     return
  end
  a=det(A);
  if lhs==2
    b=trace(A)
  end
endfunction
```

**argn** peut être aussi utilisé sous la forme **lhs=argn(1)** ou **rhs=argn(2)**. Les fonctions **varargin** et **varargout** permettent une gestion plus pointue de la liste des variables d'entrées et de sorties.

### 3.3.4 Variables globales et locales

**Scilab** fait la distinction entre les variables locales et globales au sein d'une fonction.

Les variables locales sont les variables définies comme entrantes et sortantes (out1,... in1... de la section Syntaxe), et celles qui sont affectées (déclarées) dans la fonction (hormis celles définies par le mot clef **global**).

Mais implicitement les variables du script (ou fonction appelante) se retrouvent accessibles dans la fonction appelée (mais non modifiable) à condition qu'il n'y ait pas de variable locale de même nom. Par exemple définissons une fonction qui fait appel à la variable `a` (dans un fichier : mafonc.sce) :

script Scilab
```
function  [out]=mafonc(in)
    out=a*in
endfunction
```

puis exécutons

Fenêtre Scilab
```
-->getf mafonc.sce

-->mafonc(2)
 !--error     4
undefined variable : a
at line       2 of function mafonc                   called by :
mafonc(2)

-->a=2
 a  =

    2.

-->mafonc(2)
 ans  =

    4.
```

Maintenant définissons une nouvelle fonction qui possède `a` comme variable locale, et qui appelle `mafonc`

script Scilab
```
function  [out]=mafonc2(in)
    a=-2; // variable locale
```

```
    out=mafonc(in)
endfunction
```

et maintenant

Fenêtre Scilab
```
-->getf mafonc.sce;
-->getf mafonc2.sce;
-->a=2;
-->mafonc2(2)
 ans  =

  - 4.

-->mafonc(2)
 ans  =

    4.
```

Autrement dit dans `mafonc2` la valeur de `a` est changée, dans ce cas `a` est une variable locale à la fonction. De ce fait les valeurs précédentes de `a` sont “écrasées” (localement) . A contrario, dans `mafonc`, `a` et considéré comme une variable globale (ou persistante).

Il est possible par le mot clef **global** de définir une variable globale. Cette variable est accessible pour toute fonction qui déclare la variable comme globale, redéfinissons `mafonc`

script Scilab
```
function  [out]=mafonc(in)
    global a
    out=a*in
endfunction
```

et relançons l'exécution

Fenêtre Scilab
```
-->getf mafonc.sce;
Warning :redefining function: mafonc

-->global a

-->a=2
 a  =

    2.

-->mafonc(2)
 ans  =

    4.
```

```
-->mafonc2(2)
 ans  =

    4.
```

Finissons par plusieurs mots clefs pour gérer les variables globales.

- **isglobal(**_var_**)** vrai si _var_ est déclarée "globale".
- **clearglobal(**_var_**)** supprime la variable de la liste des variables globales. un appel sans argument supprime toutes les variables globales.
- **who('global')** renvoie la liste des variables globales (homologue local **who('local')**).

## 3.4 Mise au point d'un programme

Scilab possède un jeu d'instructions de débuggage (**setbpt**, **delbpt**, **dispbpt**). L'usage n'est pas très conviviale et **Scilab** n'intègre pas d'éditeur qui permette de placer ou d'enlever à la souris de tels points d'arrêts. Mais on peut, tout simplement, placer une commande **pause** à tout endroit (même dans une fonction[7]) pour arrêter l'exécution et se retrouver sous un prompt de la forme

Fenêtre Scilab

```
-1->
```

à ce moment on a accès à toutes les variables locales. Toutes les manipulations mêmes graphiques sont possibles. Il est même envisageable d'exécuter, de cet endroit, un autre script que sera un sous processus du premier ...

La commande **return** continue l'exécution, sans tenir compte des modifications apportées durant l'arrêt. Les variables sont accessibles en lecture durant le mode pause. Si elles sont modifiées, c'est une copie "locale" qui est modifiée. Pour pouvoir continuer avec les valeurs modifiées il faut "recopier" les variables par la commande [_var1_**,** _var2_**, ...**]**=resume(**_var1_**,** _var2_**, ...**).

Lorsqu'une exécution ne répond plus, la commande **Contrôle-C** suspend l'exécution. Elle agit à tout moment comme la fonction **pause**. Par contre on ne sait pas où l'exécution c'est arrêtée, pour cela il faut utiliser la fonction **whereami()** ou [**ligne,fonc**]**=where()**.

**Remarque :** Le 1 qui apparaît dans le prompt "-1->" signifie que nous sommes dans un sous processus. Pour abandonner les processus il faut utiliser la commande **abort**.

Quand on lance un script (commande **exec**), apparaît dans l'espace de travail les lignes, une à une exécutées. Cela permet de suivre l'exécution de celui-ci. Par contre l'exécution d'une fonction est, à comparer, silencieuse. la commande **mode** permet de changer cet état.

En spécifiant

[7]Contrairement a l'instruction **setbpt**, il est nécessaire de recharger (recompiler) la ou les fonctions concernées.

- **mode(-1)** aucune ligne (hors commande **disp**) n'apparaît. C'est la valeur par défaut lors de l'appel une fonction.
- **mode(0)** c'est la règle d'affichage du point virgule qui s'applique : si une instruction est suivi de ';' alors il n'y a pas d'affichage, par contre en cas de retour à la ligne ou ',', le résultat de la commande est affiché.
- **mode(1)** ou **mode(3)** valeur par défaut pour la commande **exec**. Les lignes interprétées sont affichées (avec le résultat quand il n'y a pas de ';').

Il est donc possible de changer le "mode" dans une fonction pour avoir un regard sur ce qu'elle fait.

## 3.5 Un peu d'optimisation

Tout programmeur est confronté, à un moment ou un autre, au temps de calcul et donc à l'estimation du coût des algorithmes. **Scilab** étant en mode interprété, il faut connaître quelques comportements classiques engendrés par ce processus. Deux appels successif de la commande **timer()** permettent de mesurer le temps écoulé (le premier appel initialise le temps de référence, le second renvoie le temps écoulé).

Pour illustrer, regardons le calcul de la somme cumulée d'un vecteur, initialisons un vecteur **x**[8] :

Fenêtre Scilab
```
-->x=rand(1,400000);
```

puis calculons un temps qui sera de référence[9] :

Fenêtre Scilab
```
-->timer(); y=cumsum(x);T_ref=timer()
 T_ref  =

    0.06
```

Maintenant adoptons plusieurs stratégies pour effectuer le même tâche

Fenêtre Scilab
```
-->timer(); z(1)=x(1); ...
-->for i=2:length(x), z(i)=z(i-1)+x(i); end, ...
-->T1=timer()
 T1  =

      2179.82
```

Impossible[10] de croire ce résultat un deuxième appel pour le confirmer

[8]La taille de `x` est volontairement grande, la taille mémoire (ou de pile) nécessite d'être augmentée, **stacksize()** renvoie la valeur actuelle et permet de la modifier.

[9]On peut remarquer une certaine imprécision sur plusieurs appels successifs, avec une décroissance du temps lors du second appel, c'est le processus d'affectation (ou re-affectation) qui fait cette différence.

[10]Il est possible d'écrire la boucle sous la forme `for i=x(2 :$), z($)=z($-1)+i ; end` ce qui donne le même temps de calcul.

Fenêtre Scilab

```
...
-->T1=timer()
 T1  =

    9. 91
```

cela donne un temps beaucoup plus raisonnable. Mais dans cette deuxième exécution, `z` n'est pas une variable inconnue[11]. Elle existe et possède déjà la "bonne" taille. Dans la première exécution chaque affectations sur `z` nécessite une ré-allocation (dynamique) de la variable et engendre un travail exagérément élevé. Pour ce convaincre de l'exactitude des mesures précédentes faire

Fenêtre Scilab

```
-->clear z
```

et recommencer.

Ce mécanisme peut être enrayé par une pré-allocation :

Fenêtre Scilab

```
-->clear z

-->timer(); z=zeros(x); z(1)=x(1); ...
-->for i=2:length(x), z(i)=z(i-1)+x(i); end, ...
-->T1=timer()
 T1  =

    9.87
```

Ceci conduit à retenir deux principes :

- la ré-allocation de variable est à éviter[12].
- utiliser au mieux les fonctions vectorisées qui permettent d'éviter ou de limiter l'usage de la boucle[13] **for**.

L'écriture vectorisée engendre rapidement des mécanismes d'extraction ou d'affectation. Il est bon de se poser la question de savoir les règles à observer. Pour cela faisons un petit test. Sont proposées les deux fonctions suivantes qui calculent de deux manières distinctes la somme des éléments d'une matrice :

script Scilab

```
function out=Scol(A)
    // sommation par extraction de colonnes
```

[11]Ceci est une source d'erreur. En effet un script exécuter avec une taille de vecteur diffèrent (souvent plus petit) engendre des erreurs inconnues jusque la. Une autre forme d'erreur est le script qui fonctionne, puis une fois qu'on le relance dans un **Scilab** fraîchement ouvert apparaissent des messages inattendus. Des variables existantes dans le domaine de travail ont disparus lors de la clôture de cession

[12]Il faut faire un compromis entre le coût engendré et l'effort de programmation engendré, il est toujours possible de faire un premier "jet" non-optimal puis en second lieu améliorer.

[13]Il y a de nombreux cas où l'écriture vectorisée contracte notablement le nombre de lignes de code.

```
    [n,p]=size(A);
    som=zeros(n,1);
    for j=1:p
       som=som+A(:,j);
    end
    out=sum(som)
endfunction
```

et

script Scilab

```
function out=Slign(A)
    // sommation par extraction de lignes
    [n,p]=size(A);
    som=zeros(1,p);
    for i=1:n
       som=som+A(i,:);
    end
    out=sum(som)
endfunction
```

Après avoir chargé ces fonctions on les exécute pour obtenir le temps d'exécution :

Fenêtre Scilab

```
-->getf Scol.sce; getf Slign.sce;

-->A=rand(3000,3000);
-->timer();Scol(A),timer()
 ans  =

    4497372.4
 ans  =

    0.62
-->timer();Slign(A),timer()
 ans  =

    4497372.4
 ans  =

    2.98
```

On retrouve ici un classique des langages possédant des tableaux à double entrées, le mode de stockage détermine une façon optimale d'accéder aux éléments. En l'occurrence **Scilab** stocke les matrices colonnes par colonnes (comme le fortran) à l'instar du C/C++.

A remarquer, la possibilité de syntaxe différentes comme par exemple

script Scilab
```
function out=Scol2(A)
    // sommation par extraction de colonnes
    [n,p]=size(A);
    som=zeros(n,1);
    for col=A
       som=som+col;
    end
    out=sum(som)
endfunction
```

pour donner

Fenêtre Scilab
```
-->timer();Scol2(A),timer()
 ans  =

    4497372.4
 ans  =

    0.65
```

Cette écriture est aussi compétitive que `Scol`.

Le ratio entre les deux modes d'accès est ici de 4.8. Il est faible comparé aux ratios obtenus dans le cas de réaffectation ou de comparaisons avec la boucle **for**, mais il est clairement non-négligeable.

Il nous faut un outil de "profiling" pour détecter les lignes gourmandes. Ceci existe avec les commandes **profile**, **showprofile** et **plotprofile**. Pour cela il faut charger la fonction, que l'on veut examiner, sous un mode "profile" par

Fenêtre Scilab
```
-->getf('Scol.sci','p');
```

puis faire un appel de cette fonction et examiner le profile

Fenêtre Scilab
```
-->Scol(A);

-->showprofile('Scol')
function out=fun(A)                 |1   |0   |0|
                                    |1   |0   |0|
[n, p] = size(A);                   |1   |0   |4|
som = zeros(n, 1);                  |1   |0   |4|
for j = 1:p,                        |3000|0.01|0|
  som = som + A(:, j);              |3000|0.62|7|
end                                 |1   |0   |0|
```

```
out = sum(som);                     |1    |0    |3|
endfunction                         |1    |0    |0|
```

La première colonne indique le nombre d'exécution de la ligne (depuis le dernier **getf( ... ,'p')**), la seconde le temps cumulé, et la troisième l'effort d'interprétation. Cette dernière est à interpréter avec modération, ne signifiant pas forcément un sur-coût.

C'est l'outil idéal pour traquer, et faire maigrir le temps d'exécution.

## 3.6 Conclusions diverses

Comme promis, **Scilab** peut être considéré et utilisé comme un langage de programmation. Il faut rapidement éviter les écueils d'une écriture non-vectorisée. Pour ce qui est de la programmation, il est du ressort de chacun de développer son style. Bien sûr de simples règles comme : commenter largement ses lignes, ou d'utiliser des noms de variables le plus explicite ou usuelle dans leur rôle sont à appliquer (un compteur d'indice est souvent `i`, `j` ou `k`, appeler une matrice `i` ... a vos risques et périls !).

Pour les personnes se sentant démunies face à leur page blanche, je propose un schéma simple. Il ne faut pas oublié que **Scilab** est un interpréteur de commande possédant des ressources de traitement ou d'affichage de données remarquable. Il serait dommage de se priver d'une tel manne.

Le modèle proposé pour le développement d'une (petite) application est le suivant : il est basé sur l'ouverture simultané de deux fichiers.

le premier `MesFonctions.sci` contient l'ensemble des fonctions qui vont être écrites :

script Scilab
```
// Fichier de fonctions : MesFonctions.sci
//---------------------------------------
function out=premierefonc(in)
    ....
endfunction

function out=deusiemefonc(in)
    ....
endfunction
     .
     .
     .
```

Le second fichier `main.sce` est le fichier qui sera exécuté, en premier. Il est possible (voir recommander) de compiler les fonction par la fonction **getf**

script Scilab
```
// Ceci est le scrip principal
//---------------------------

getf MesFonctions.sci //a partir de la les fonctions sont chargées
```

```
// maintenant on peut faire ce que l'on a à faire
...
plot2d(sol,u)
...
```

Ainsi pour lancer le programme il suffit de lancer la commande

Fenêtre Scilab
```
-->exec main.sce
```

et automatiquement les dernières modifications (même dans le fichier de fonctions) sont prises en compte. De plus une fois le script exécuté, restent les variables disponibles pour être consultées ou affichées. Au programmeur de prendre ses dispositions pour avoir la possibilité à cet endroit, de post-traiter ces résultats.

Une habitude nouvelle, à comparer avec d'autre langages, et de pouvoir tester la syntaxe immédiatement. Supposons qu'une malencontreuse erreur de syntaxe se soit glisser dans une fonction, générant immanquablement une erreur à l'exécution. Immédiatement informé de la ligne correspondante, le programmeur insère un **pause** ou **setbpt** avant de relancer l'exécution pour se retrouver en mode "pause" avant d'avoir commis la faute. A ce moment là, il est recommander de tester diverses syntaxes et de regarder explicitement les variables concernée par la ligne fautive.

Sur ces dernières recommandations, il peut être sage de faire les quelques exercices proposés dans la section suivante.

## 3.7 Exercices

Les premiers exercices portent sur des mécanismes standards de phénomènes d'erreurs d'arrondi. Cette section ne fait intervenir que des algorithmes basics, néanmoins les mécanismes mis en avant doivent être en tête pour toute personne voulant faire du calcul numérique à précision finie ! (impérativement à connaître).

**Exercice I :** Somme d'une suite alternée.
Le développement en série entière de la fonction cosinus, nous donne la formule approchée :

$$\cos(x) \simeq \sum_{k=0}^{n} (-1)^k \, \frac{x^{2k}}{(2k)!}$$

I.a) A l'aide de la formule précédente, écrire un script qui calcule la valeur approchée de cosinus pour une valeur donnée par l'utilisateur.
On sera particulièrement attentif à deux choses :
- le factoriel aux dénominateur empêche de prendre des valeurs de $n$ trop grandes bien que le terme général de la suite tende vers 0. Il faut équilibrer le calcul de $\frac{x^{2k}}{(2k)!}$ en décomposant celui-ci sous une forme $\frac{x^{2k}}{(2k)!} = \frac{x^2}{1\times 2}\frac{x^2}{3\times 4}\cdots\frac{x^2}{(2k-1)\times 2k}$.
- du fait que le terme générale tende vers 0, on prendra comme critère de choix de $n$ le moment où le terme générale ne contribue plus à la somme.

Calculer $\cos(39)$ (réponse : $\cos(39) \simeq 0.26664293235994$).

I.b) Tracer suivant $0 \leq k \leq n$, le terme $\frac{x^{2k}}{(2k)!}$ pour le calcul de $\cos(39)$. Expliquer l'erreur constatée.

I.c) Toujours à l'aide de la série entière, trouver et justifier une méthode pour calculer avec plus de précision (autant que possible) $\cos(39)$. On comparera le nombre de décimales exactes par rapport à la question a).
Indication : on peut éventuellement utiliser la valeur de $\pi$ (mot clef **Scilab** : %pi).

Il est possible de construire de nombreux exemples de ce type, avec entre autre, le développement en série entière de $e^{-x}$ pour $x > 0$. L'astuce pour le calcul et de transformer le calcul de la suite alternée en une suite à termes positifs $e^{-x} = \frac{1}{e^x}$

**Exercice II :**

Soit la famille d'intégrales :

$$I_n = \int_0^1 \frac{x^n}{10+x}\, dx \quad n \in I\!N$$

II.a) Montrer que $I_0 = \ln \frac{11}{10}$ et que l'on a la récurrence $I_n = \frac{1}{n} - 10 I_{n-1}$.

II.b) Montrer l'approximation :

$$\frac{1}{11(n+1)} \leq I_n \leq \frac{1}{10(n+1)} \quad n \in I\!N^*$$

II.c) Écrire un programme qui calcule, à partir de la relation de récurrence initialisée avec $I_0$ : $I_5$, $I_{10}$ et $I_{30}$. Vérifier les inégalités précédentes et expliquer ce qui ce passe.

II.d) Même question que c) mais par récursion inverse (décroissante), en partant de l'approximation $I_n \simeq \frac{1}{10(n+1)}$ pour n suffisamment grand. Calculer $I_0$ ($I_0 = 0.095310179804$) $I_5$, $I_{10}$ et $I_{30}$. Analyser et commenter.

Les exemples sont multiples et illimités comme : $J_n = \int_1^e \ln(x)^n\, dx \quad n \in I\!N$...

# Chapitre 4

# Graphiques

Un tel environnement ne peut avoir de sens sans la possibilité de sorties graphiques. **Scilab** possède un jeu suffisamment étendu de fonctions graphiques 2d et 3d. De plus il permet l'appel à des fonctions d'affichage de bas niveau qui, le cas échéant, peuvent permettre de concocter des rendus de toute sorte.

## 4.1 Affichage 2d : commande plot

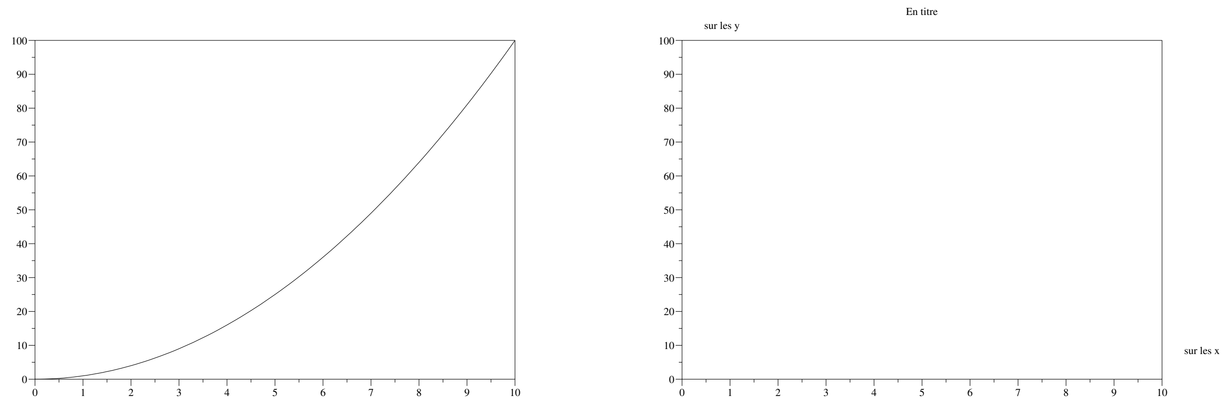



FIG. 4.1 – Sortie simple avec ou sans titre

La fonction **plot** est la plus simple et la moins paramétrable des fonctions graphiques. Sa syntaxe est de la forme :

Fenêtre Scilab

```
-->x=0:0.1:10;
-->plot(x.^2)
-->plot(x,x.^2) // rendu identique a la premiere syntaxe
-->plot(x,x.^2,'sur les x','sur les y','En titre')
```

Cette fonction sera délaissée ou seulement utilisée pour avoir un affichage rapide en ligne du tracé d'un tableau de valeurs.

## 4.2 Fenêtre graphique et commandes génériques

L'appel de la fonction **plot** a ouvert une fenêtre graphique faisant apparaître quatre nouveaux boutons

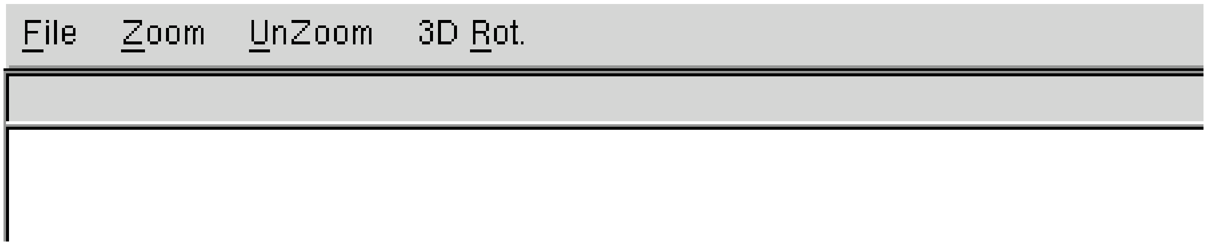



FIG. 4.2 – Menu de la fenêtre graphique

- 3D Rot. Ce bouton a une signification pour des graphiques 3D.
- Zoom et UnZoom un clic sur ce premier bouton, et la fenêtre graphique est en attente d'un premier clic du bouton gauche, suivi d'un second pour déterminer la zone rectangulaire à agrandir. Il est possible de répéter l'opération, le second bouton remettant dans l'état initial.
- File ce bouton ouvre un menu :
  - Clear Efface le(s) graphiques.
  - Select Non documenté.
  - Print Imprime le graphique.
  - Export Renvoi sur une boite de dialogue, il est possible d'exporter l'image du graphique sous format gif (format standard compatible windows), PostScript (en-capsulé où pas), Xfig (logiciel de dessin vectoriel) ou Latex (PostScript en-capsulé + Latex).
  - Save Sauvegarde le graphique dans un format propre a **Scilab**.
  - Load Charge un graphique **Scilab** préalablement sauvegarder.
  - Close Ferme la fenêtre graphique.

Il est tout à fait possible d'ouvrir plusieurs fenêtres simultanément. Par défaut, lors d'un appel à une fonction graphique (**plot** ...), la fenêtre n°0 est ouverte. La commande **xset('window',*num*)** permet d'ouvrir et/ou de basculer dans différentes fenêtres.

Fenêtre Scilab

```
-->x=0:0.1:2*%pi;
-->plot(x,exp(cos(x))) // affichage dans le fenêtre courante
-->xset('window',1)     // ouvre la fenêtre 1
-->plot(x,cos(exp(x))) // affichage dans le fenêtre 1
-->xset('window',0      // rétabli le fenêtre 0 comme fenêtre courante
-->xdel(1)              // ferme la fenêtre 1
```

Apparaît la commande **xdel**[1] qui ferme la fenêtre graphique.

Pour effacer le contenu d'un fenêtre taper **xclear(*num*)**[2] ou **xbasc(*num*)**.

## 4.3 La commande plot2d

Cette fonction, avec ces différentes options, sera d'un usage courant et immodéré pour grand nombre d'applications.

En premier lieu il est possible comme dans le cas de la fonction plot de ne spécifier qu'un tableau de valeurs ou un couple abscisses ordonnées **plot2d(x,y)**[3]

Fenêtre Scilab
```
-->x=0:0.1:10;
-->plot2d(x.^2)
-->plot2d(x,x.^2)
```

Néanmoins, sous cette forme on peut afficher plusieurs courbes simultanément, pour cela il suffit de passer en argument une matrice en lieu et place des abscisses et ordonnées ; les colonnes sont les valeurs à tracer :

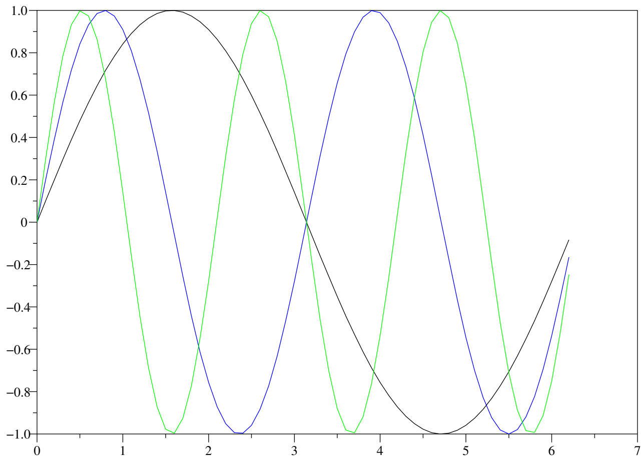



FIG. 4.3 – `sin(x),sin(2x),sin(3x)` sur $[0, 2\pi]$

Fenêtre Scilab
```
--> x=[0:0.1:2*%pi]';
--> plot2d(x,[sin(x) sin(2*x) sin(3*x)])
```

[1]Spécifiée sans arguments : **xdel()** c'est la fenêtre courante qui est fermée. **xdel(1 :2)** efface les fenêtre 1 et 2.

[2]Même remarque que pour **xdel**, **xclear()** efface la fenêtre courante.

[3]En fait, est pris en considérations les points de coordonnées `[x,y]`, un segment de droite est tracé du premier point jusqu'au dernier. Pour s'en convaincre `x=0 :0.1 :2*%pi ; plot2d(cos(x'),sin(x'))`.

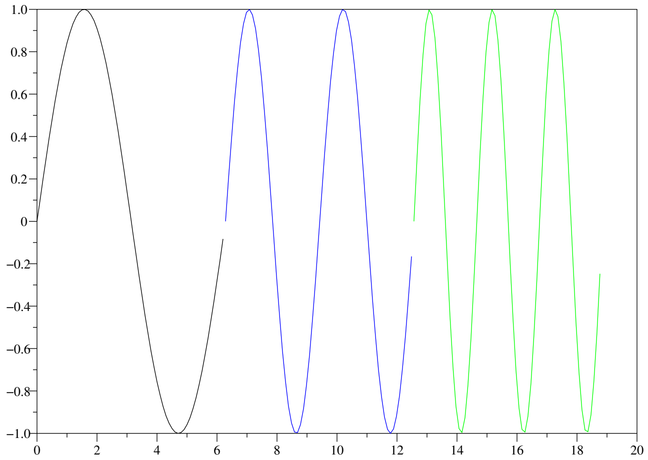



FIG. 4.4 – `sin(x),sin(2x),sin(3x)` sur $[0, 2\pi]$, $[2\pi, 4\pi]$, $[4\pi, 6\pi]$

ou encore

Fenêtre Scilab
```
--> x=[0:0.1:2*%pi]';xx=[x,x+2*%pi,x+4*%pi];
--> plot2d(xx,[sin(x) sin(2*x) sin(3*x)])
```

La règle étant : soit, comme dans ce cas, les abscisses sont représentées par un vecteur colonne et donc commun à l'ensemble des ordonnées (fig 4.3), soit il y a autant de colonnes en abscisses et en ordonnées, et les couples abscisses ordonnées sont affichés (fig 4.4).

**plot2d** offre la possibilité de définir un certain nombre d'attributs que peuvent être la couleur du trait, continu ou pointillé, les valeurs maximales et minimales ... Toutes ces options possèdent une syntaxe de la forme

**plot2d(x,y,***nom_option1***=valeur1,***nom_option2***=valeur2...)**

Dans la suite, est donné un aperçu de quelques options utiles de **plot2d**.

**Choix de la couleur et du trait :** *nom_option* : `style=...`

La couleur ou le mode de tracé (en continu ou point) est géré par le nom d'option **style**. C'est un tableau ligne d'entiers, avec un attribut pour chaque courbe, i.e. de taille le nombre de colonnes des ordonnées. Il spécifie la couleur (par défaut 1,2,...nombre de colonnes de "y"). Une valeur positive correspond à une couleur (4.6[4]) et une valeur négative ou nulle remplace l'affichage continu par un symbole (voir la table de correspondance 4.5).

Voici un exemple de commande (voir résultat figure 4.7)

[4]Ceci est la table de couleur par défaut mais peut être modifiée avec la commande **colormap**, pour obtenir la table de couleur courante faire **getcolor()** (figure 4.6).

| style | 0 | -1 | -2 | -3 | -4 | -5 | -6 | -7 | -8 | -9 |
|---|---|---|---|---|---|---|---|---|---|---|
| mark | . | + | × | ⊕ | ♦ | ♢ | △ | ▽ | ♣ | ○ |

FIG. 4.5 – Tableau de correspondances pour “style” négatif

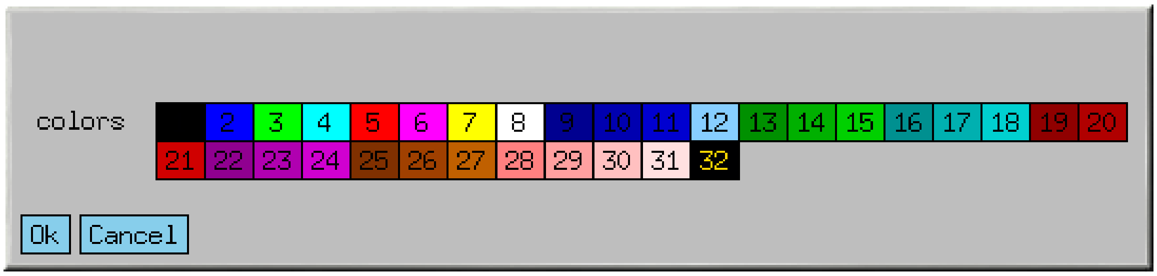



FIG. 4.6 – Table de correspondances des couleurs : **getcolor()**

Fenêtre Scilab
```
--> x=[0:0.1:2*%pi]';
--> plot2d(x,[sin(x) cos(x)],style=[-7,14])
```

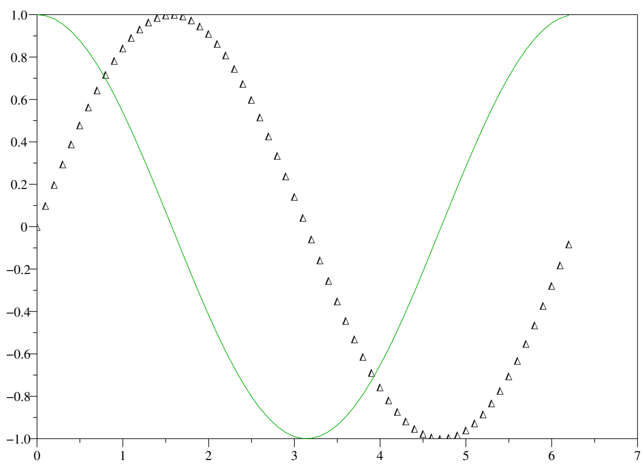



FIG. 4.7 – Utilisation de **plot2d** avec option “style=[-7,14]”

Il est tout à fait possible de passer le style en argument implicite[5] en faisant

Fenêtre Scilab
```
--> plot2d(x,[sin(x) cos(x)],[-7,14])
```

[5]Ceci correspond à une ancienne syntaxe qui imposait d’avoir les arguments des options dans un ordre précis.

le tableau **style** doit impérativement être en 3ème position.

**Choix de la zone de donnée ou "range" :** *nom_option* : `rect=...`

A chaque appel de **plot2d** les valeurs maximales et minimales sont évaluées pour offrir un affichage qui permet de voir la courbe en entier. Dans le cas de plusieurs appels successifs de la commande **plot2d**, les courbes sont superposées mais de plus, en cas de changement de valeurs extrêmes, les courbes sont ré-affichées pour permettre une visualisation simultanée (voir exemple qui suit)

Fenêtre Scilab
```
--> x1=[0:0.1:2*%pi]';
--> x2=[-5:0.1:5]';
--> plot2d(x1,sin(x1),1);
--> plot2d(x2,exp(x2),2);
```

Néanmoins en spécifiant une option de la forme **rect=[xmin,ymin,xmax,ymax]**, sont imposées les coordonnées de la fenêtre d'affichage. Ceci peut permettre par exemple un "zoom" sur une partie de la courbe[6].

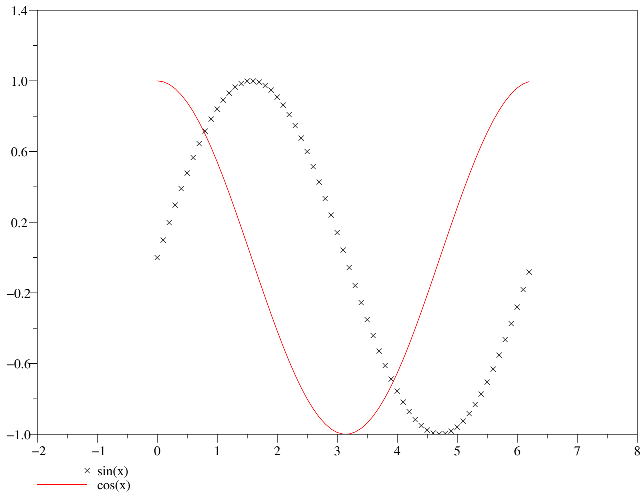



FIG. 4.8 – plot2d(x,[sin(x) cos(x)],leg='sin(x)@cos(x)',rect=[-2,-1,8,1.3],style=[-2,5])

**Nommer les courbes :** *nom_option* : `leg=...`

Il est agréable de lire explicitement sur un graphique rempli de courbes qui correspond à quoi. Pour cela, il est possible d'utiliser l'option **leg=chaîne de caractères** pour voir apparaître en bas à droite de la courbe un petit trait de couleur ou le symbole de la courbe suivi de sa nomination. Dans le cas de plusieurs courbes, est passé en argument de **leg** une seule chaîne de caractère avec les différents champs séparés par un "@".

[6] Il est tout aussi possible d'utiliser le bouton Zoom de la fenêtre graphique.

Fenêtre Scilab
```
--> x=[0:0.1:2*%pi];
--> plot2d(x,[sin(x) cos(x)],leg='sin(x)@cos(x)');
```

**Graduer les axes :** *nom_option* : `nax=...`

Il est possible de choisir le nombre de graduation et de sous graduation par axe en spécifiant **nax=[nx,Nx,ny,Ny]**, Nx et Ny représentent le nombre de graduation, tandis que nx et ny celui de la sous graduation (entre chaque graduation).

**Échelles logarithmiques :**

Pour obtenir un affichage suivant une échelle logarithmique sur l'axe vertical ou horizontal, il suffit de rajouter dans l'appel de **plot2d** deux caractères prenant les valeurs "n" pour normal et "l" pour logarithme de la manière suivante

Fenêtre Scilab
```
-->x=logspace(1,3,10);
-->plot2d('ln',x,log(x)) \\ axe x en echelle log, y echelle normale
-->plot2d('ll',x,x.^2)   \\ axes x et y en echelle log
```

**Options avancées :** *nom_option* : `strf='xyz'`

Il y a de multiples façon d'afficher un graphique, en recalculant les échelles pour obtenir a courbe entière, afficher les axes, graduation isométrique... Ces attributs ont une réelle importance si l'on cherche à afficher sur le même système d'axes plusieurs courbes ou parties de courbe possédant des caractéristiques différentes comme le nombre de point par exemple.

Pour introduire l'utilité de l'option `strf` dans **plot2d** partons sur le dernier exemple cité, avec deux tableaux de dimension différentes, de tel sorte qu'un appel de **plot2d** ne peux suffire

Fenêtre Scilab
```
-->x0=linspace(0,3,30);
-->x1=linspace(1,2,20);
-->plot2d(x0,sin(x0))  // premier appel
-->plot2d(x1,cos(x1),3,strf='000')//second sans changer d'axes
```

D'une manière générale et systématique `strf='xyz'` avec la table de correspondance

| x | |
|---|---|
| 0 | Pas de légende |
| 1 | Légendes affichées (voir `leg`) |

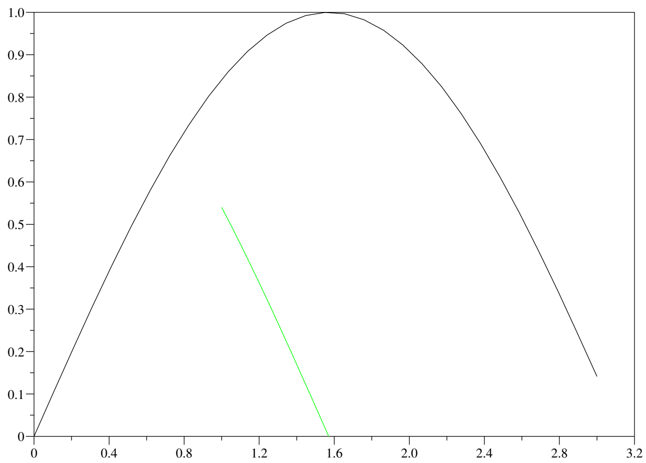



FIG. 4.9 – Utilisation de `strf='000'` pour superposer deux courbes

| y | |
|---|---|
| 0 | Utilisation des bornes et échelle du dernier appel de **plot2d** |
| 1 | Échelle standard, bornes spécifiées par `rect` |
| 2 | Échelle standard, bornes calculées suivant les valeurs de x et y |
| 3 | Échelle isométrique, bornes spécifiées par `rect` |
| 4 | Échelle isométrique, bornes calculées suivant les valeurs de x et y |
| 5 | Graduation élargie, bornes spécifiées par `rect` |
| 6 | Graduation élargie, bornes calculées suivant les valeurs de x et y |

| z | |
|---|---|
| 0 | rien n'est dessiné autour du dessin |
| 1 | les axes sont dessinés, l'axe y est dessiné à gauche. |
| 2 | le dessin est entouré d'une boîte sans graduations. |
| 3 | les axes sont dessinés, l'axe y est dessiné à droite |
| 4 | les axes sont dessinés au milieu du cadre |
| 5 | les axes sont dessinés de manière à se croiser au point (0,0).<br>Si ce point (0,0) ne se situe pas dans le cadre, les axes n'apparaissent pas. |

## 4.4 Légende

Avec les options de **plot2d** il est possible de faire apparaître le nom des courbes, mais il existe une possibilité souvent plus esthétique c'est la commande **legends**

Fenêtre Scilab
```
-->t=0:0.1:2*%pi;
-->plot2d(t,[cos(t'),sin(t')],[-1,2]);
```

```
-->legends(["sin(t)";"cos(t)"],[-1,2])
```

La syntaxe générale est `legends(`*nom_courbes*,*couleur*,*position*`)`, avec
- *nom_courbes* un tableau verticale des chaînes de caractères correspondants aux labels voulus.
- *couleur* tableau correspondant à l'option `style` de **plot2d** pour le choix des couleurs ou marques (taille identique à *nom_courbes*).
- *position*
  si cette option n'est pas spécifiée, alors le cadre est positionné à l'aide de la souris
  si *position* est un vecteur `[x,y]`, ce dernier représente le coin en haut à gauche du cadre de légende.
  si *position* est un entier, alors : 1 = en haut à droite, 2 en haut à gauche, 3 en bas à gauche, 4 en bas à droite et 5 spécifie le placement interactif avec la souris.

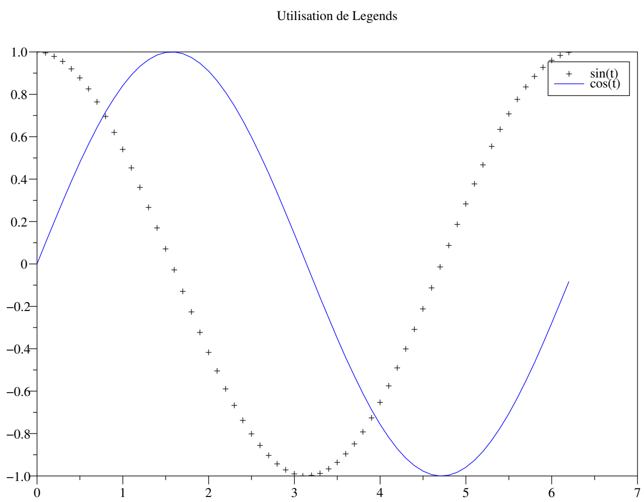



FIG. 4.10 – Mise en valeur avec **legends** et **xtitle**.

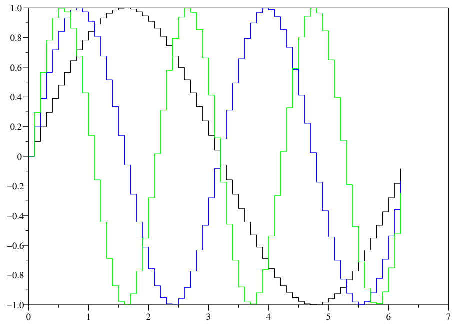
La commande **xtitle** permet d'enjoliver les sorties graphiques par l'ajout d'un titre et d'intitulés d'axes.

```
Fenêtre Scilab
-->xtitle('Utilisation de Legends')
```

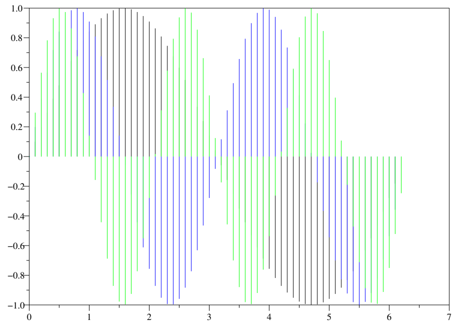
## 4.5 Autre primitives graphiques 2d

Il existe trois variantes[7] de **plot2d**, chacune obéissant aux mêmes règles dans le choix des options.

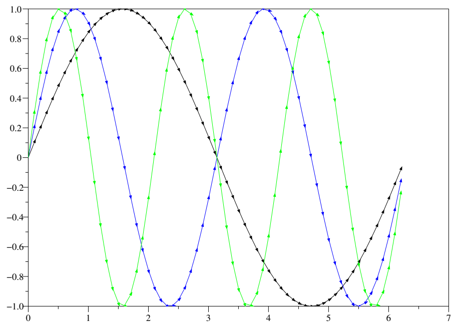
[7]**plot2d1** existe mais correspond au cas d'échelle logarithmique



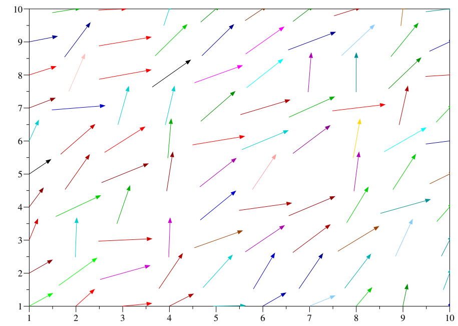

`plot2d2` fonction escalier



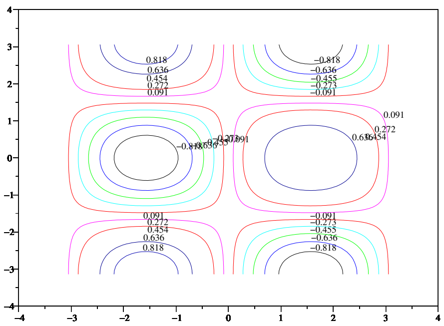

`plot2d3` barres verticales



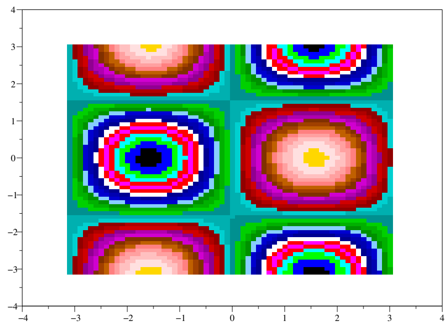

`plot2d3` parcours fléché




`champ1` champ de vecteur (colorés)




`contour2d` de $sin(x)cos(y)$

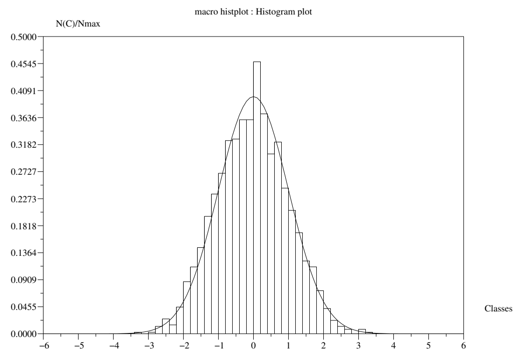



`grayplot` de $sin(x)cos(y)$

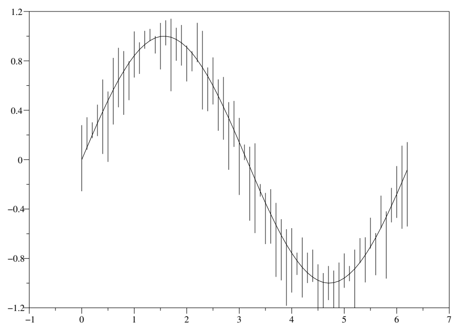
- **plot2d2** Le tracé se fait sur chaque intervalle par une constante, la valeur de gauche de l'intervalle.
- **plot2d3** En chaque point du graphique est tracé une barre verticale jusqu'à l'ordonnée 0.
- **plot2d4** La courbe est tracée en continu avec une flèche dans le sens de parcours.
- **fplot2d** Tracé d'une courbe définie par une fonction.

Il existe différentes fonctions d'affichage avec pour chacun une syntaxe particulière, le lecteur avisé consultera le manuel des fonctions correspondantes. En voici une énumération avec une visualisation dans les différentes variantes.

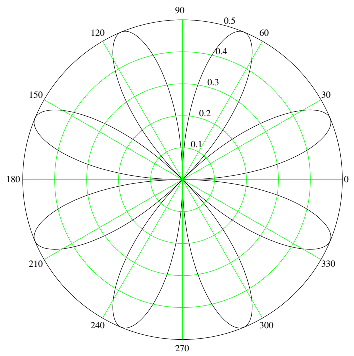
- **histplot** Tracé d'histogramme.
- **errbar** Affichage de barres verticales d'erreurs.
- **champ**, **champ1** Trace un champ de vecteur en spécifiant les coordonnées du point de départ et celles du vecteur : `champ(x,y,fx,fy)`.
- **contour2d**, **fcontour2dcontourf** Trace les lignes isocontours d'une matrice de valeurs.
- **grayplot**, **fgrayplot**, **Sgrayplot**, **Sfgrayplot** Trace par niveaux de couleur une matrice de valeurs.
- **Matplot**, **Matplot1** Affiche en 2d des carreaux (ou bandes) de couleur donné par la partie entière des valeurs de la matrice.



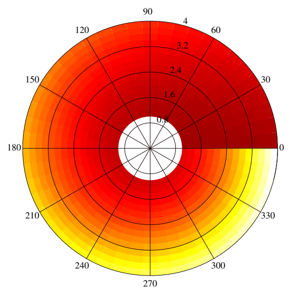

`histplot` histogramme



`errbar` barres verticales d'erreurs



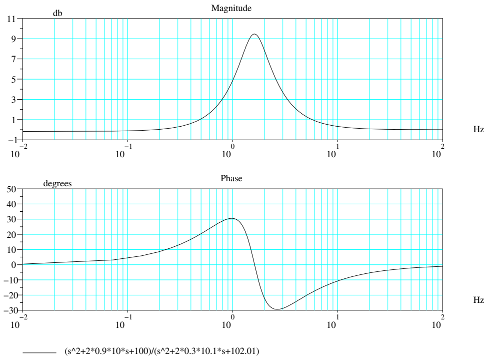
`polarplot` Tracé en coordonnées polaires



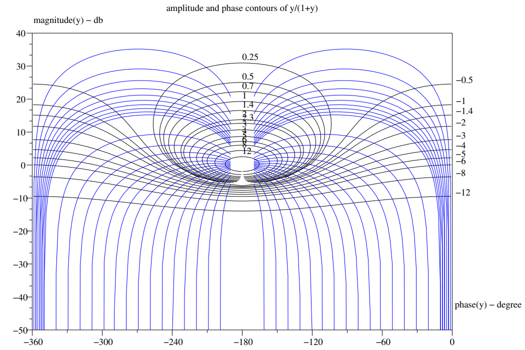
`graypolarplot` équivalent de `grayplot`

Il existe également des fonctions d'affichage spécifique utilisé en théorie du signal et automatisme[8]

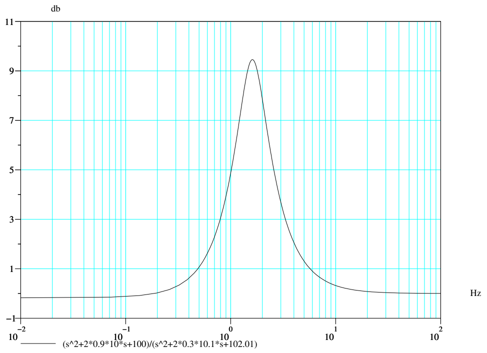
[8] A vérifié !



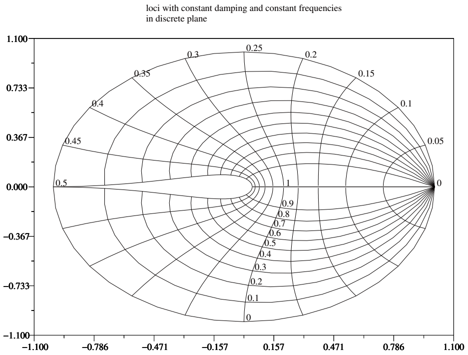

`bode`




`chart`



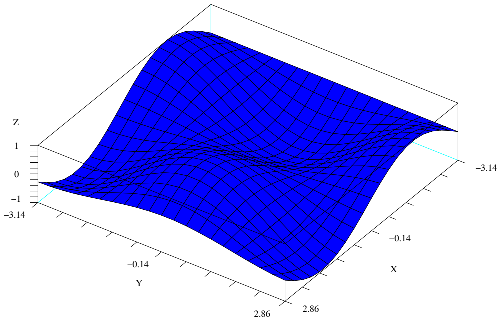

`gainplot`



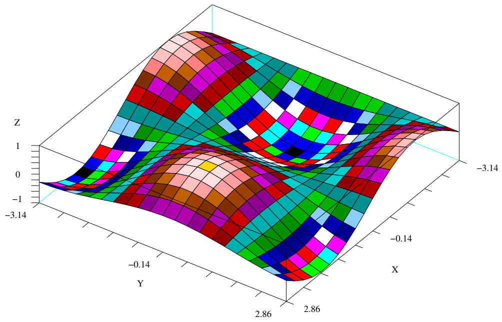

`zgrid`

## 4.6 Affichage 3d

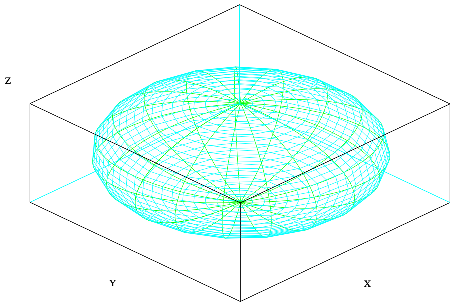
La commande générique d'affichage tridimensionnelle est la fonction **plot3d**.



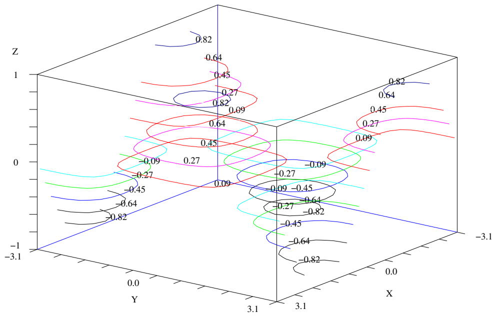

plot3d



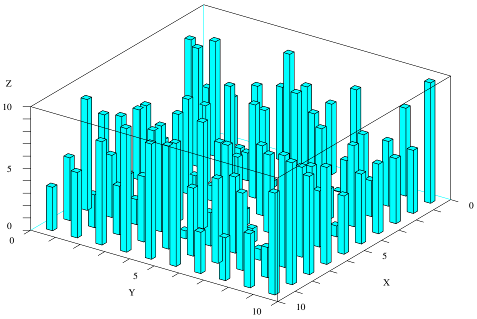

plot3d1



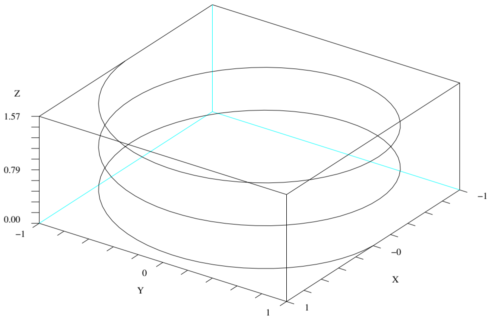

plot3d3




contour




hist3d




param3d

# Chapitre 5

# Solutions des exercices

## 5.1 Exercices section 2.2.6

1. res = 0.
2. res = 0.
3. `prod(1 :5)`.
4. `v=1 ./(1 :10)`.
5. `u=rand(1 :10) ;u=u/sqrt(u*u')` ou `u=rand(1 :10) ;u=u/sqrt(u.^2)`.
6. `2.^(1 :8)`.
7. `v=sin((1 :20)*2*%pi/20)`.
8. `v=v($ :-1 :1)`.
9. `u=1 :0.2 :3`.
10. `u(5 :5 :$)`.

## 5.2 Exercices section 2.3.6

1. `A=2*eye(10,10)` ou `A=diag(2*ones(1 :10))`.
2. `B=A($ :-1 :1, :)` ou `B=A( :,$ :-1 :1)`.
3. `C=rand(3,4)`
   - `C( :,[4 2 3 1]) ;`
   - `C(1 :3,1 :3)`
   - `[C ;C]`.
4. - `A=[ones(3,3), (2 :4)' ; 1 :4]`
   - `A=matrix(1 :9,3,3)` ou `A=cumsum(ones(3,3))`
   - `A=ones(3,3) ;A([1,3],[1,3])=1`
   - `A=(1 :4)' ;A=A( :,ones(1,4))`
5. `A=rand(4,4) ;B=rand(4,4) ;C=max(A,B)`.

## 5.3 Exercices section 2.4.5

1. `A=rand(5,5) ;B=rand(A) ; M=A>B`.
2. `A=rand(5,5) ; A((A>0.3)&(A<0.7))`.
3. `A=rand(1,10) ;A(A>0.5)=A(A>0.5)-1`
4. `A=round(10*rand(5,5)) ;find(A==0)`

## 5.4 Exercices section 2.5.4

1. `p=poly([1 1 3 -1],’x’,’c’)`
2. `q=poly(1 :3,’x’), roots(q)`
3. `n=25 ; q=poly(1 :n,’x’), roots(q)`
4. `f=poly([0 1 -2],’x’,’c’)/poly([-1,1],’x’)`
   - `df=derivat(f)`
   - `y=horner(df,[0 2 :4])`
   - `pol=roots(df(’den’))`

## 5.5 Exercices section 2.6.5

1. `strcat(string(1 :5),’,’)`
2. `strindex(’bonjour’,’o’),strsubst(’bonjour’,’o’,’-’)`
3. `’pi=’+string(%pi)`