Petit Guide de Survie en Scilab

Romain JOLY

Institut Fourier, Université Grenoble I

Le but de ce petit guide n'est pas d'apprendre Scilab, mais plutôt de fournir d'une part une aide à l'apprentissage, d'autre part un formulaire auquel on peut se référer en cas d'oubli. Pour ce faire, il contient un peu d'explications, des petits exemples et des encadrés rappelant les commandes usuelles. Il est bien entendu conseillé d'utiliser ce guide devant Scilab, de tester les commandes et de lire l'aide en ligne : c'est en essayant qu'on apprend le mieux.

1 Introduction

Scilab est un logiciel de calcul numérique développé par l'INRIA. Il est entièrement libre et gratuit, contrairement au logiciel commercial Matlab. On peut le télécharger par exemple à l'adresse
`http://www.scilab.org/download/index_download.php?page=release.html`
Contrairement à Maple, Scilab n'est pas un logiciel de calcul formel ou exact, mais un logiciel de calcul approché. Il permet de faire très rapidement des calculs matriciels, résoudre des systèmes linéaires, simuler des équations différentielles, etc. Notez que Scilab fait partie des logiciels disponibles aux épreuves de l'agrégation.
Pour lancer Scilab sous Linux : ouvrir un terminal et taper simplement `scilab &`. Le symbole `&` permet de lancer le logiciel tout en gardant la possibilité de continuer à taper des commandes dans la fenêtre Linux. On obtient alors une fenêtre Scilab dans laquelle on peut taper des lignes de commandes.
Trois façons de travailler en Scilab :
• On tape directement les commandes dans la fenêtre Scilab principale. L'avantage est bien sûr la simplicité d'utilisation. Toutefois, dès que l'on souhaite écrire un programme de plus d'une ligne, cela devient vite trop compliqué. De plus, si l'on veut relancer des lignes de commandes, il faut retaper les lignes une à une.

```
                Copyright (c) 1989-2004
            Consortium Scilab (INRIA, ENPC)
        -----------------------------------------

 Startup execution:
  loading initial environment

-->2*3-1
 ans  =

    5.
```

• On écrit les commandes dans un fichier à l'aide d'un traitement de texte, puis on l'exécute dans Scilab avec la commande `exec`. Cette méthode est indispensable dès que l'on souhaite faire un programme un peu long ou que l'on veut pouvoir modifier facilement des paramètres du programme sans avoir à tout retaper. Les dernières versions de Scilab contiennent un traitement de texte intégré que l'on lance en cliquant sur le bouton `Editor` (si la version est plus ancienne, on passe par un traitement de texte extérieur). Exemple : on tape dans le fichier `2*3-1`, on sauvegarde sous le nom `test.sci`, et on tape dans la fenêtre principale de Scilab :

```
-->exec('test.sci');
 ans  =

    5.
```

NB : les guillemets servent à encadrer une chaîne de caractères (ici un nom de fichier), le point-virgule signifie que le résultat de la commande (ici l'appel du programme) ne doit pas s'afficher. Sans le point-virgule, Scilab réécrit toutes les commandes du programme :

```
-->exec('test.sci')

-->2*3-1
 ans  =

    5.
```

• Troisième possibilité : on écrit dans un fichier séparé une fonction, par exemple

```
function x=calcul()
x=2*3-1;
endfunction;
```

On sauvegarde le fichier sous le nom `test.sci`, et on tape dans la fenêtre Scilab `getf('test.sci');`. Maintenant, Scilab connait une nouvelle fonction baptisée `calcul` que l'on peut appeler :

```
-->calcul
 ans  =

    5.
```

L'avantage est que l'on peut faire dépendre la fonction de plusieurs paramètres (voir plus loin) et que l'on peut mettre plusieurs fonctions dans un même fichier (par exemple plusieurs exercices d'un même TD). Par contre, le formalisme est un peu plus lourd qu'un simple programme lancé par `exec` et il ne faut pas oublier `getf` à chaque fois que l'on modifie le fichier `.sci`.

Comment quitter ? Aller dans le menu `File` et cliquer sur `quit` ou bien taper la commande `quit`. Si cela ne fait rien, retenter. En cas d'échec, on peut taper `exit` ou choisir `kill` dans le même menu `File` jusqu'à arrêt du programme. Il peut y avoir des difficultés pour quitter si des commandes tournent en boucle ou si des programmes ont planté. Attention, il ne faut jamais se déloguer sans quitter Scilab. En effet, dans le cas contraire, il est possible que Scilab continue à tourner même après la fin de la session et cela tant que personne ne le tuera.

A l'aide : pour ouvrir le menu d'aide, taper `help;` dans Scilab. Pour obtenir de l'aide sur une commande précise, taper `help` suivi de la commande (par exemple `help exec;`). En cas de problème, l'aide en ligne se trouve à l'adresse :

`http://www.scilab.org/product/man-eng/index.html`

`help;`	ouvre le menu d'aide
`help` *commande* `;`	ouvre l'aide à propos de *commande*
`quit; exit; kill;`	quitte la session, quitte Scilab, tue le programme en cours
↑ ↓	rappelle les lignes de commandes précédentes

Exercice 1 : Calculer $425 * 6^2$. Refaire le même calcul en passant par un programme et la commande `exec`. Quitter les logiciels.

2 Syntaxe de base

Ponctuation : une commande doit toujours se terminer par une virgule, un point-virgule, un retour de ligne ou deux petits points. Si la commande se termine par une virgule ou un retour, le résultat de la commande sera affiché (la virgule permet de mettre plusieurs commandes dans une même ligne). Si la commande se termine par un point-virgule, elle sera effectuée mais le résultat ne sera pas affiché. Les deux petits points servent à mettre une commande sur plusieurs lignes.

```
-->2/4, 6+1; 7*2
 ans  =

    0.5
 ans  =

    14.

-->1+2+3+4+5+..
-->6+7+8+9
 ans  =

    45.
```

En pratique, on utilise le point-virgule pour les calculs intermédiaires et on ne met pas de point-virgule pour afficher le résultat final. On évite aussi de mettre une commande sur plusieurs lignes ou trop de commandes par ligne pour des raisons de lisibilité.
NB : Dans un programme, on prendra l'habitude de mettre un point virgule de façon systématique après chaque commande. En effet, un simple oubli de point-virgule dans une boucle peut entraîner l'affichage de plusieurs centaines de lignes.

Les guillemets servent à délimiter une chaîne de caractères :

```
-->'bonjour',"bonsoir"
 ans  =

 bonjour
 ans  =

 bonsoir
```

On peut aussi utiliser le double slash `//` pour mettre des commentaires. Tout ce qui suit sur cette ligne ne sera pas considéré par Scilab. C'est très pratique pour mettre des com-

mentaires dans un programme.

```
-->2*3 //blablabla !!*$$'
 ans  =

    6.
```

Variables : Scilab est conçu pour manipuler les matrices, et a fortiori les scalaires qui sont des matrices 1*1. On peut affecter des valeurs à des variables en utilisant le signe `=`. Un nom de variable est un mot composé de lettres et de chiffres et qui commence par une lettre. Notez que Scilab fait la différence entre les majuscules et minuscules. On peut ainsi retenir un résultat et l'utiliser plus tard.

```
-->a1=3;a2=4;A1=5;a1+A1*a2
 ans  =

    23.
```

Remarquez que, grâce aux points-virgules, seul le dernier résultat s'affiche. Le résultat du dernier calcul est mis dans la variable `ans`. On peut donc le réutiliser, ou bien l'enregistrer dans une variable.

```
-->b=ans*2+1
 b  =

    47.
```

On peut aussi auto-appeler une variable, par exemple pour augmenter un compteur.

```
-->i=1; i=i+1; i=i+1; i=i+1
 i  =

    4.
```

`,  ;`	affiche ou n'affiche pas le résultat
`a=1`	donne la valeur 1 à la variable a
`//`	permet de mettre un commentaire
`..`	permet d'écrire sur plusieurs lignes
`clear`	réinitialise toutes les variables
`'bonjour'`	une chaîne de caractères

Nombres et fonctions usuelles : Scilab affiche des nombres à virgule fixe tant qu'il est facile de les lire, sinon il utilise des nombres à virgule flottante. Ainsi, il écrit `0.05` pour $0,05$ et `5.792E+85` pour $5,792 * 10^{85}$. Voici une liste de commandes qui permettent de faire avec Scilab toutes les opérations classiques d'une calculatrice.

`%pi %e %i`	les constantes π, e et $i = \sqrt{-1}$
`abs sign`	la valeur absolue (ou le module) et la fonction signe
`real imag`	la partie réelle et la partie imaginaire
`exp log log10`	l'exponentielle, le logarithme népérien et le log en base 10
`cos sin tan cotg`	cosinus, sinus, tangente et cotangente
`acos asin atan`	arccosinus, arcsinus et arctangente
`cosh sinh tanh`	les mêmes en hyperboliques
`acosh asinh atanh`	...
`sqrt`	la racine carrée
`floor`	la partie entière (plus grand entier inférieur)
`round`	le plus proche entier
`ceil`	la partie entière plus un
`int`	la partie entière anglaise (`floor` si $x > 0$, `ceil` sinon)
`rand()`	un nombre au hasard entre 0 et 1

Exercice 2 : Calculer $\cos(\pi/3)$, $e^{i\pi} + 1$ et la partie entière de la racine de 39. Additionner les trois nombres obtenus.

Exercice 3 : Résoudre l'équation $\sinh(x) = \ln 7$. Résoudre l'équation $z^2 = 1 + i$.

Exercice 4 : Tirer un entier aléatoire entre 0 et 9.

3 Calcul matriciel

Gestion des vecteurs et matrices : dans Scilab, on utilise les crochets pour délimiter une matrice, les virgules pour séparer les composantes d'une ligne et des points virgules pour séparer les colonnes.

```
-->A=[1,2,3;4,6,7]
 A  =

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

```
-->B=[8;9]
 B  =

!   8. !
!   9. !
```

On obtient l'élément $A_{i,j}$ par la commande `A(i,j)`. On peut remplacer un nombre i ou j par le caractère `$` pour obtenir le dernier élément. On peut aussi extraire la matrice qui est l'intersection des lignes $i1$ et $i2$ et des colonnes $j1$ et $j2$ par la commande `A(i1:i2,j1:j2)`. Si on ne spécifie pas $i1$ et $i2$, Scilab considère qu'il faut prendre toute la ligne ou colonne.

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

    1.
 ans  =

    7.
 ans  =

!   2. !
!   6. !
 ans  =

!   1.    2. !
!   4.    6. !
```

Notez que pour un vecteur (ligne ou colonne), un seul chiffre suffit : les commandes `B(2,1)` et `B(2)` sont équivalentes.

`[1,2,3], [1;2;3]`	un vecteur ligne et un vecteur colonne
`[1,2;3,4]`	une matrice 2*2
`A(i,j)`	l'élément A_{ij}
`A(:,5), A(3:7,2:4)`	des matrices extraites
`w($), w($-1)`	les derniers et avant-derniers éléments d'un vecteur w

Construction des matrices : On peut modifier les éléments d'une matrice par le signe `=` comme pour une variable. On peut faire de même pour des matrices extraites.

```
-->B(1,1)=0, A(:,2)=[1;1]
```

```
 B  =

!   0. !
!   9. !
 A  =

!   1.    1.    3. !
!   4.    1.    7. !
```

En général, on manipule très rapidement de grandes matrices (par exemple 100*100). Pour définir de telles matrices, on ne rentre pas la valeur de chaque coefficient, mais on utilise des matrices prédéfinies comme `ones`, `zeros`, `eye` ou `toeplitz` .

```
-->ones(2,3), zeros (4,1), eye(4,4)
 ans  =

!   1.    1.    1. !
!   1.    1.    1. !
 ans  =

!   0. !
!   0. !
!   0. !
!   0. !
 ans  =

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

-->toeplitz([1,2,3,0,0])
 ans  =

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

Dans Scilab, on utilise souvent des vecteurs constitués de nombres régulièrement espacés. Ce type de vecteur s'obtient grâce au symbole `:`. Ainsi `1:10` est la liste des entiers

de 1 à 10. Par défaut, le pas entre deux coefficients est 1. Si on veut le changer, il suffit de mettre le pas au milieu. Par exemple, `1:0.5:5` est la liste des demi-entiers de 1 à 5.

```
-->1:10, 1:0.5:5, -2:4, 2:2:9
 ans  =

!   1.    2.    3.    4.    5.    6.    7.    8.    9.    10. !
 ans  =

!   1.    1.5    2.    2.5    3.    3.5    4.    4.5    5. !
 ans  =

! - 2.  - 1.    0.    1.    2.    3.    4. !
 ans  =

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

`min:pas:max`	un vecteur avec les nombres de min à max espacés de pas
`linspace(min,max,n)`	n nombres régulièrement espacés entre min et max
`zeros(i,j)`	une matrice remplie de 0 de taille i*j
`zeros(A)`	une matrice remplie de 0 de même taille que A
`ones(i,j)`	une matrice remplie de 1
`eye(i,j)`	une matrice avec des 1 sur la diagonale, des 0 ailleurs
`toeplitz(v)`	une matrice symétrique basée sur le vecteur `v`
`rand(i,j)`	une matrice i*j remplie de nombres aléatoires entre 0 et 1
`diag(u)`	une matrice diagonale avec comme coeffs ceux du vecteur u
`diag(A)`	l'extraction du vecteur dont les coeffs sont la diagonale de A
`A(3:7,2:4)=B`	redéfinition d'une sous-matrice de A
`[A,B], [A;B]`	concaténation horizontale et verticale de matrices
`A(i:j,:)=[]`	suppression des lignes i à j

Opérations sur les matrices : dans Scilab, si on applique une fonction définie sur les scalaires sur une matrice, on obtient la matrice dont chaque coefficient est l'image du coefficient original.

```
-->(1:5)^2, cos([0,%pi/3,%pi])
 ans  =

!   1.    4.    9.    16.    25. !
 ans  =

!   1.    0.5  - 1. !
```

Si on a deux matrices, on peut les sommer, les soustraire etc. Attention : `A*B` désigne le produit matriciel entre A et B alors que `A.*B` désigne le produit terme à terme. Il faut bien sûr que les tailles des matrices soient compatibles.

```
-->[1,1;2,2]+[1,2;-1,-2], [1,1;2,2].*[1,2;-1,-2], [1,1;2,2]*[1,2;-1,-2]
 ans  =

!   2.    3. !
!   1.    0. !
 ans  =

!   1.    2. !
! - 2.  - 4. !
 ans  =

!   0.    0. !
!   0.    0. !

-->[1,2;3,4]*[1;-1], [1,2;3,4]*[1,-1]
 ans  =

! - 1. !
! - 1. !
                                    !--error    10
inconsistent multiplication
```

NB : se méfier de la syntaxe `./` ou `.*`. En effet, `1./A` sera compris comme $1,00000 * A^{-1}$. Pour obtenir la matrice dont les coefficients sont les inverses de ceux de A, il faut écrire par exemple `(1)./A`.
Etant donnée une matrice A, on obtient le conjugué de la transposée ${}^t\overline{A}$ en utilisant l'apostrophe.

```
-->[1,2;3,4]', [1,2,3+%i]'
 ans  =

!   1.    3. !
!   2.    4. !
 ans  =

!   1.       !
!   2.       !
!   3. - i   !
```

En outre, Scilab sait calculer le déterminant d'une matrice ainsi que son inverse de façon très efficace. On notera aussi l'existence de la commande `sum` qui somme tous les coefficients d'une matrice et de la commande `norm` qui donne la racine carrée de la somme des carrés des coefficients.

`A*B A^2`	multiplication matricielle et puissance d'une matrice
`A+B, A-B, A.*B, A./B, A.^B`	opérations terme à terme
`A', u'`	conjugué de la transposée d'une matrice, d'un vecteur
`sin(A), exp(A)`	calcule le sinus et l'exponentielle de chaque coef
`size(A)`	nombre de lignes et de colonnes de A
`det(A), rank(A)`	le déterminant de A et son rang
`inv(A)`	l'inverse de A
`sum(u), prod(u)`	la somme et le produit des coefs d'un vecteur
`max(u), min(u)`	le plus grand et le plus petit coef d'un vecteur ou d'une matrice
`[m,xmin]=min(u)`	renvoie le minimum d'un vecteur ainsi que la coordonnée où celui-ci est atteint
`norm(u), norm(u,1)`	les normes ℓ^2 et ℓ^1 d'un vecteur

Spectre : Scilab possède des commandes permettant de trouver très simplement le spectre d'une matrice. Ainsi la commande `spec(A)` donne la liste des valeurs propres de A. Noter que l'on peut aussi obtenir les vecteurs propres. Ceux-ci sont stockés dans une matrice que l'on peut récupérer en demandant explicitement que la commande `spec(A)` sorte la liste de valeurs propres ainsi que la matrice des vecteurs propres.

```
-->A=[1,2;3,2];spec(A)
 ans  =

! - 1. !
!   4. !

-->[V,l]=spec(A)
 l  =

! - 1.    0  !
!   0     4. !
 V  =

! - 0.7071068  - 0.5547002 !
!   0.7071068  - 0.8320503 !
```

```
-->A*V(:,1)
 ans  =

!   0.7071068 !
! - 0.7071068 !
```

Une commande très similaire est la commande `bdiag` qui permet de diagonaliser une matrice. Sans spécification, Scilab renvoie la matrice diagonalisée et si l'on demande deux sorties, on obtient en outre la matrice de passage. En fait, la commande donne aussi une décomposition de type Jordan pour les matrices non-diagonalisables.

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

! - 1.    0. !
!   0.    4. !

-->[D,P]=bdiag(A)
 P  =

! - 0.7071068  - 0.5656854 !
!   0.7071068  - 0.8485281 !
 D  =

! - 1.    0. !
!   0.    4. !

-->A=[1,-1;4,5]; bdiag(A)
 ans  =

!   3.  - 4. !
!   0.    3. !
```

`spec(A)`	la liste des valeurs propres de A
`[V,l]=spec(A)`	la liste des valeurs propres et des vecteurs propres de A
`bdiag(A)`	la décomposition de Jordan de A

Exercice 5 : Construire les matrices suivantes en utilisant le moins d'opérations possible.

$$A = \begin{pmatrix} 1 & 2 \\ 9 & 7 \end{pmatrix} \quad B = \begin{pmatrix} 2 & 0 & 1 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{pmatrix} \quad C = \begin{pmatrix} 1 & 1 & 1 \\ 2 & 1 & 2 \\ 1 & 1 & 1 \end{pmatrix} .$$

Extraire la dernière ligne de A. Extraire la matrice à l'intersection des deux premières lignes et colonnes de C.

Exercice 6 : Construire la matrice de taille 100*100 de la forme suivante.

$$\begin{pmatrix} 2 & 1 & 0 & \cdots & 0 \\ 0 & 2 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ \vdots & & \ddots & \ddots & 1 \\ 0 & \cdots & \cdots & 0 & 2 \end{pmatrix}$$

Exercice 7 : Calculer la somme des entiers de 1 jusqu'à 100, calculer 100!. Donner une façon de calculer toutes les valeurs du cosinus sur les entiers de 1 à 100 le plus rapidement possible.

Exercice 8 : Calculer les normes ℓ^2 et ℓ^1 d'un vecteur sans passer par la commande `norm`.

Exercice 9 : Résoudre le système linéaire suivant

$$\begin{cases} x - y &= 5 \\ -2x + y &= -7 \end{cases}$$

4 Fonctions

Scilab est surtout fait pour manipuler des matrices et des vecteurs. En général, une fonction sera donc une liste de valeurs. Par exemple, si on veut manipuler la fonction cosinus sur $[0, 1]$, pour faire un graphique ou pour simuler un phénomène, on manipulera plutôt un vecteur du type `u=cos(0:0.01:1)`. Ainsi, pour un graphique classique, on donnera à la fonction `plot2d` des vecteurs de nombres plutôt que des fonctions. Notez qu'en général, les données de simulations réelles ne sont effectivement qu'une liste de nombres (par exemple la température mesurée toutes les secondes).
Toutefois, on peut avoir besoin de définir une fonction précise afin de l'utiliser plusieurs fois dans un programme ou pour rendre la programmation plus claire. La façon la plus simple de faire est d'utiliser la commande `deff`.

```
-->deff('y=f(x)','y=(cos(5*x))^2')
```

```
-->f(0:4)
 ans  =

!   1.    0.0804642    0.7040410    0.5771257    0.1665310 !
```

La première variable de la commande `'y=f(x)'` est une chaîne de caractères précisant le nom de la fonction, le nom des variables d'entrée ainsi que le nom des variables de sortie. La seconde partie `'y=(cos(5*x))^2'` est une liste de commandes définissant la valeur des variables de sortie.
Si on veut que la fonction sorte plusieurs valeurs, il faut les récupérer quand on appelle la fonction. Sinon, seule la première sortie de la fonction est affichée, comme on le note ci-dessous.

```
-->deff('[y1,y2]=g(x1,x2)','y1=2*x2;y2=y1+x1;');

-->g(1,2)
 ans  =

    4.

-->[x,y]=g(1,2)
 y  =

    5.
 x  =

    4.
```

Dans le cas d'une liste d'instructions vraiment longue, on utilisera plutôt la commande `function y=f(x)`, une liste de commandes, `endfunction;`. On peut ainsi écrire la fonction dans un fichier `toto.sci` à l'aide d'un traitement de texte (que l'on lance par exemple en cliquant sur `Editor` dans la fenêtre Scilab). Puis on appelle la fonction dans Scilab grâce à la commande `getf` (méthode similaire à la commande `exec`). Par exemple, pour définir la fonction g précédente, on peut ouvrir un fichier `fonction-g.sci` dans lequel on écrit :

```
function [y1,y2]=g(x1,x2)

y1=2*x2;  //En plus je peux mettre des commentaires
y2=y1+x1; //pour me rappeler le sens de chaque ligne

endfunction;
```

Ensuite, on appelle la fonction dans Scilab et on peut l'utiliser comme avant.

```
-->getf('fonction-g.sci');

-->[x,y]=g(1,2)
 y  =

    5.
 x  =

    4.
```

NB : Si on envisage d'appliquer la fonction sur des coordonnées de vecteurs ou de matrices, il est bon de prendre le formalisme des opérations termes à termes des matrices. Ainsi, on remplacera par exemple `a*b` par `a.*b`. En outre, on notera que l'on peut mettre plusieurs fonctions dans un même fichier tant qu'elles sont bien séparées par la commande `endfunction;`.

Scilab peut aussi calculer des intégrales. La commande `integrate` utilise une fonction définie par une chaîne de caractères, `intg` utilise une fonction externe déjà définie.

```
-->integrate('cos(x)','x',0,1)
 ans  =

    0.8414710

-->deff('y=f(x)','y=cos(x)'); intg(0,1,f)
 ans  =

    0.8414710
```

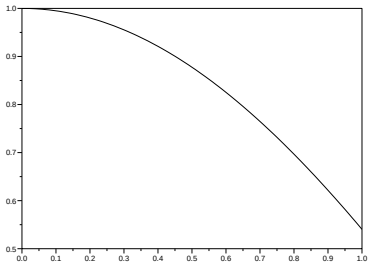
`deff('y=f(x)','y=x^2')`	appelle f la fonction $x \mapsto x^2$
`function endfunction;`	définit une fonction longue ou à stocker dans un fichier `.sci`
`getf`	récupère une fonction définie dans un fichier
`integrate, intg`	pour calculer des intégrales

5 Graphiques

Graphique classique : la commande la plus usuelle est la commande `plot2d` qui demande deux vecteurs lignes de même longueur. Le premier vecteur correspond aux ab-

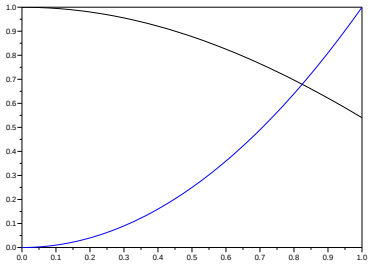
scisses (en général, une liste de points régulièrement espacés), le second correspond aux ordonnées (les données, les valeurs de la fonctions sur les points du premier vecteur, etc.). La commande `plot2d` ne fait ensuite que tracer les segments reliant les points successifs.

```
-->x=0:0.01:1; y=cos(x);

-->plot2d(x,y)
```



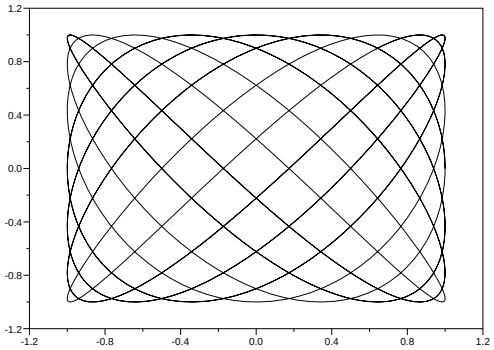
Si on retrace un graphique, la deuxième courbe se superposera à la précédente. Pour nettoyer le graphique, on utilise la commande `clf()`. On peut aussi afficher plusieurs graphiques en donnant directement des matrices $k * n$, où k est le nombre de courbes.

```
-->x=0:0.01:1; y1=cos(x); y2=x^2;

-->plot2d([x;x]',[y1;y2]');
```



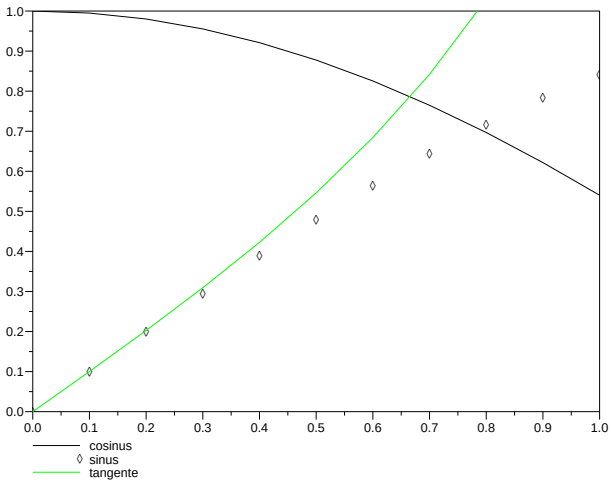
Par défaut, Scilab affiche une fenêtre contenant tout le graphique et seulement le graphique. On peut imposer la fenêtre d'affichage en utilisant l'option `rect=[xmin,ymin,xmax,ymax]`. Pour afficher la courbe dans le carré $[-1, 1]\times[0, 2]$, on utilisera donc `plot2d(x,y,rect=[-1,0,1,2])`. Notez que Scilab trace en fait la courbe passant par la liste de points de coordonnées (x_i, y_i). Pour un graphe de fonction, les x_i doivent être ordonnés par ordre croissant, mais Scilab fait aussi le graphe si l'ordre est quelconque. Cela permet par exemple de tracer des courbes paramétrées.

```
-->clf()

-->t=0:0.01:10; x=cos(7*t); y=sin(9*t);

-->plot2d(x,y,rect=[-1.2,-1.2,1.2,1.2])
```



Il existe d'autres options possibles. Par exemple, `plot2d(x,y,style=2)` donne la couleur de la courbe. Un style négatif trace simplement la liste des points sans les relier (cela permet par exemple de placer quelques points particuliers sur un graphique). Quand plusieurs courbes sont tracées, une légende est utile pour les distinguer. On utilise l'option `leg='leg1@leg2@...'` où `leg`i est la légende de la courbe i.

```
-->clf(); x=0:0.1:1;

-->plot2d([x;x;x]',..
-->[cos(x);sin(x);tan(x)]',..
-->style=[1,-5,3],rect=[0,0,1,1],
-->leg='cosinus@sinus@tangente');
```

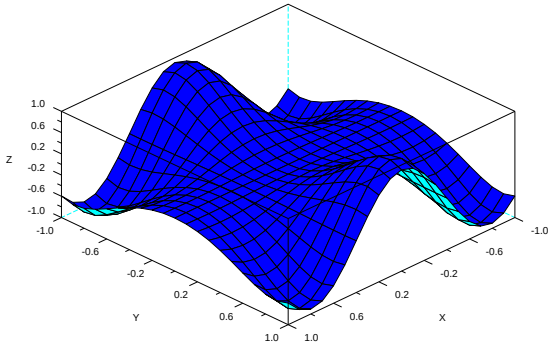


Pour tracer plusieurs graphiques, on peut utiliser la commande `subplot(n,m,k)`. Celle-ci découpe la fenêtre graphique en une matrice $n \times m$ de sous-graphiques et place le graphique suivant à la k-ième place. En faisant la même opération pour différents k, on peut tracer plusieurs graphiques en parallèle.
Scilab peut aussi ouvrir plusieurs fenêtres graphiques. Par défaut, la fenêtre est numérotée 0. Si on veut travailler dans la fenêtre 1, on tape `xset('window',1)` et on travaille alors dans la fenêtre 1. Pour revenir sur la fenêtre 0, il suffit de taper `xset('window',0)`.

`plot2d(x,y)`	dessine un graphique
`rect=[xmin,ymin,xmax,ymax]`	option pour choisir la taille de la fenêtre
`style=2`	option donnant la couleur de la courbe
`style=-5`	option pour tracer une série de points
`clf()  xbasc()`	efface les graphiques précédents
`subplot(n,m,k)`	découpe la fenêtre en sous-graphiques
`xset('window',1)`	change la fenêtre graphique utilisée

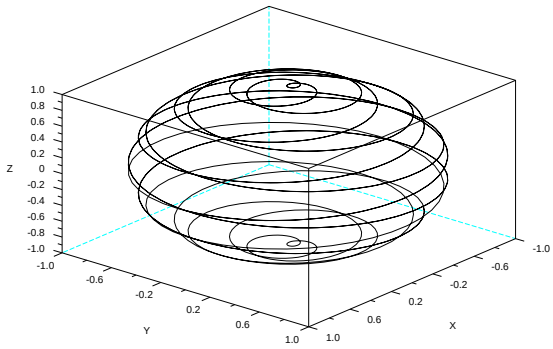
Autres graphiques : Scilab possède un grand nombre de commandes graphiques. Pour toutes les voir, on peut aller sur l'aide en ligne. A part `plot2d`, les commandes les plus utiles sont les suivantes.
La commande `fplot3d` permet de tracer une surface définie par une fonction.

```
-->deff('z=f(x,y)',..
-->'z=cos(4*x)*sin(2*y^2)');

-->xbasc();

-->x=-1:0.1:1;

-->fplot3d(x,x,f);
```

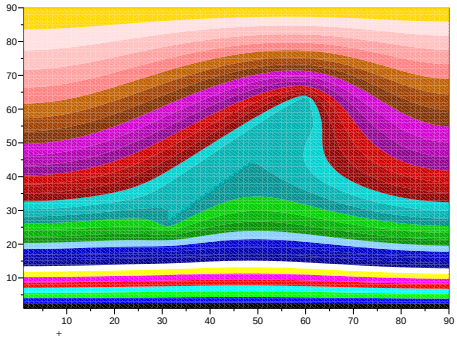


La commande `plot3d` est semblable mais cette fois la surface n'est pas définie par une fonction mais par une matrice de valeurs A_{ij} correspondants à la hauteur de la surface au point (x_i, y_j).
La commande `param3d(x,y,z)` trace une courbe passant successivement par les points de coordonnées (x_i, y_i, z_i). Cela permet donc de tracer des courbes paramétrées.

```
-->xbasc();

--> t=0:0.01:10;

--> param3d(cos(t).*cos(15*t),..
-->cos(t).*sin(15*t),sin(t));
```



Si x et y sont deux vecteurs et si A est une matrice, la commande `Sgrayplot(x,y,A)` colorie les niveaux de la surface $z(x_i, y_j) = A_{ij}$. Cela permet de tracer des graphiques représentant par exemple une répartition de pression d'un fluide ou une autre valeur dépendant d'une variable spatiale.



`fplot3d(x,y,f)` et `plot3d(x,y,z)`	trace une surface définie par une fonction f ou une matrice z
`fplot3d1(x,y,f)` et `plot3d1(x,y,z)`	idem mais avec des niveaux de couleurs
`contour(x,y,z,n)`	trace n lignes de niveau de la surface
`Sgrayplot(x,y,z)`	coloriage suivant les niveaux de la surface
`param3d(x,y,z)`	une courbe paramétrée 3D
`Matplot(A)`	dessine une grille dont chaque case (i,j) a la couleur du coefficient A_{ij} (qui doit être entier positif)

Animations : si on simule une équation aux dérivées partielles, ou si l'on souhaite tracer au fur et à mesure la trajectoire d'une équation différentielle ordinaire, on a besoin de réaliser une animation. Pour cela, il suffit simplement d'afficher un graphique à chaque pas de temps puis de l'effacer. Les deux gros problèmes qui peuvent arriver sont les suivants : le graphique scintille et on ne voit pas grand chose, ou bien le programme est très ralenti. Pour améliorer le résultat, voici quelques astuces :
- n'effacer le graphique précédent que juste avant d'afficher le nouveau,
- utiliser la commande `xpause(t);` pour afficher le graphique un peu plus longtemps (Scilab attend t microsecondes),
- ne pas afficher le graphique à chaque pas de temps, mais seulement tous les n pas en utilisant un test de division du type `if (i/n)==round(i/n) then`,
- ne pas garder en mémoire les résultats des calculs précédents qui ont déjà été affichés,
- si on trace une trajectoire d'une équation différentielle, on peut ne pas effacer le graphique à chaque fois et seulement tracer la courbe entre le temps t et $t + dt$.

Exercice 10 : Tracer la courbe de la fonction $x \mapsto x^3$ sur $[-1, 1]$. Tracer la courbe de la fonction $x \mapsto 1/x$ sur $]0, 3]$ avec un pas d'ordre 10^{-4}. La tronquer de façon à rendre le graphique lisible.

Exercice 11 : Définir la fonction $f : x \mapsto \sin(5x^2)$. Tracer son graphe.

Exercice 12 : Refaire l'exercice précédent en définissant f dans un fichier.

6 Polynômes

ATTENTION : Scilab n'est pas un logiciel de calcul formel. Il n'est donc pas fait pour

gérer correctement des calculs formels avec des polynômes. Par expérience, les calculs avec des polynômes de degré trop grand peuvent donner des résultats complètement faux. Il faut donc s'en méfier.

Un polynôme n'est rien d'autre qu'une liste de coefficients. De ce fait, il peut être facilement géré en Scilab. Toutefois, la multiplication de polynômes n'est pas forcément facile à programmer. On peut alors utiliser le type polynôme géré par Scilab. Le polynôme élémentaire de Scilab est `%s`. On peut définir directement un polynôme :

```
-->P=1+%s+4*%s^3
 P  =

              3
    1 + s + 4s
-->P^2
 ans  =

              2    3    4     6
    1 + 2s + s + 8s + 8s + 16s
```

On peut aussi définir un polynôme par la liste de ses racines grâce à `poly`. Si on veut le définir par la liste de ses coefficients, on utilise l'option `'coeff'`.

```
-->Q=poly([0,-2],'s'), R=poly([1,2,3],'s','coeff'),Q+R
 Q  =

          2
    2s + s
 R  =

               2
    1 + 2s + 3s
 ans  =

               2
    1 + 4s + 4s
```

NB : la spécification `'s'` dans la commande `poly` signifie que R est un polynôme de la variable s. On peut choisir d'autres variables. Toutefois, la multiplication de polynômes de variables différentes est impossible. Donc, soit on utilise toujours la variable s qui correspond à la variable Scilab `%s`, soit on utilise la variable x et on définit le polynôme élémentaire en posant `x=poly(0,'x');` .

```
-->x=poly(0,'x'), S=1+x+x^2, S^2
 x  =

    x
 S  =

              2
    1 + x + x
 ans  =

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

-->P*S
    !--error     4
undefined variable : %p_m_p
```

Scilab gère de même les fractions rationnelles.

```
-->(%s+1)/(%s-1)+%s
 ans  =

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

`a1+a2*%s+a3*%s^2`	un polynôme de degré deux
`poly(u,'s')`	le polynôme unitaire dont les zéros sont les coeffs du vecteur u
`poly(u,'s','coeff')`	le polynôme dont les coefficients sont ceux du vecteur u
`poly(u,'s','c')`	idem en plus court
`coeff(P)`	la liste des coefficients de P
`roots(P)`	la liste des racines de P
`factors(P)`	la liste des facteurs irréductibles de P
`horner(P,2)`	renvoie la valeur $P(2)$
`x=poly(0,'x');`	une astuce pour travailler avec des polynômes en x

Exercice 13 : Ecrire une fonction qui renvoie le degré d'un polynôme.

Exercice 14 : Ecrire une fonction qui associe à un polynôme son polynôme dérivé.

7 Solveurs

Résolution d'équations : Pour résoudre un système d'équations linéaires $Ax = b$, il suffit bien sûr d'utiliser par exemple la commande `inv(A)*b`. Si on a affaire à des équations non-linéaires, il faut utiliser d'autres méthodes comme la méthode de Newton. Scilab possède des commandes permettant d'obtenir directement le résultat (il s'agit d'un résultat approché). Voici par exemple la résolution du système $\begin{cases} x^2 = 1 \\ x - y = 2 \end{cases}$ à l'aide de la commande `fsolve` :

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

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

!   1.  - 1. !
```

La première donnée de `fsolve` est un point de départ de l'algorithme. Changer cette valeur peut permettre de trouver différentes solutions.

```
 -->fsolve([-5,0],f)
 ans  =

! - 1.  - 3. !
```

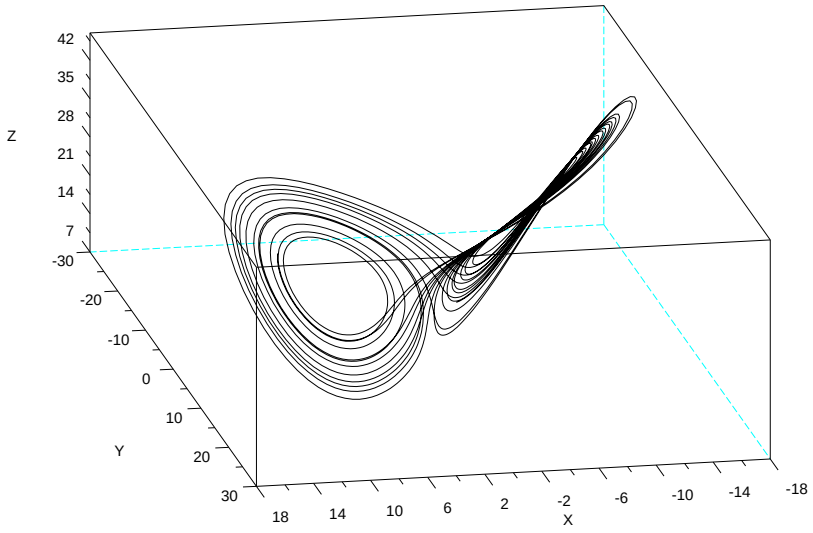
Simulation d'équations différentielles : Pour tracer les solutions de l'équation différentielle

$$\begin{cases} y'(t) = f(t, y(t)) \quad t \in [t_0, t_{max}] \\ y(t_0) = y_0 \end{cases}$$

on utilise la commande `ode(y0,t0,T,f)` où T est un vecteur de valeurs du temps décrivant l'intervalle $[t_0, t_{max}]$. Les points de T donnent la discrétisation : plus T est fin, plus la simulation sera précise. La commande `ode(y0,t0,T,f)` renvoie un vecteur correspondant aux valeurs de $y(t)$ sur les temps donnés dans T. On peut ensuite tracer la courbe avec les commandes graphiques appropriées.
Voici un exemple simulant le système d'équations de Lorenz

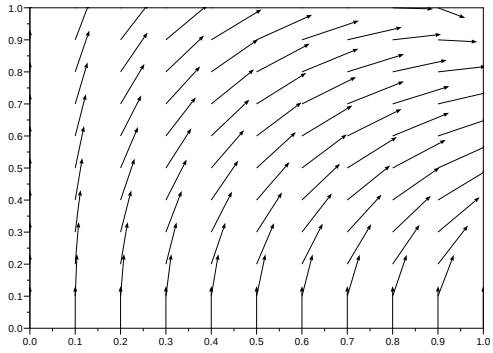
$$\begin{cases} \frac{dx}{dt} = 10(y - x) \\ \frac{dy}{dt} = -xz + \rho x - y \\ \frac{dz}{dt} = xy - \frac{8}{3}z \end{cases}$$

```
-->deff('y=f(t,x)','y=[10*(x(2)-x(1)),-x(1)*x(3)+28*x(1)-x(2),..
-->x(1)*x(2)-8/3*x(3) ]');

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

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



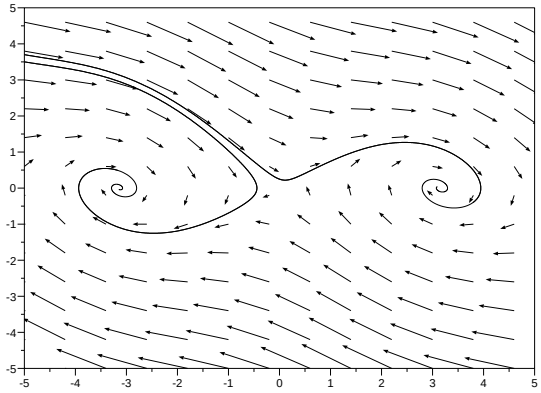
Afin d'illustrer le comportement qualitatif d'une équation différentielle ordinaire, il peut être intéressant d'afficher le champ de vecteurs associé. Pour cela, deux possibilités. La première est la commande `champ(x,y,Fx,Fy)` qui trace un champ de vecteurs donné par deux matrices F_x et F_y correspondant aux valeurs des coordonnées du champ aux différents points du graphique.

```
-->xbasc();

-->x=0:0.1:1; A=x'*x;

-->Fx=sin(3*A); Fy=cos(2*A);

-->champ(x,x,Fx,Fy);
```



La deuxième possibilité est la commande `fchamp(f,t0,x,y)` qui trace le champ de vecteurs associé à l'équation différentielle $y'(t) = f(t, y(t))$ au temps t_0. Attention : la fonction f doit avoir comme variables d'entrée t et y, même si elle ne dépend pas de t. Voici un exemple où sont superposés le champ de vecteurs et quelques trajectoires de l'équation différentielle.

```
-->deff('y=f(t,x)','y=[x(2),sin(x(1))-0.5*x(2)]');

-->x=-5:0.8:5; xbasc();

-->fchamp(f,0,x,x);

-->y=ode([-5;3.5],0,0:0.01:20,f);

-->plot2d(y(1,:),y(2,:));

-->y=ode([-5;3.7],0,0:0.01:20,f);

-->plot2d(y(1,:),y(2,:));
```



`fsolve(x0,f)`	résoud le système $f(x) = 0$.
`ode(y0,t0,T,f)`	donne la solution de l'équa. diff. $y'(t) = f(t, y(t))$, $y(t_0) = y_0$
`champ(x,y,fx,fy)`	trace le champ de vecteurs $(f_x(x, y), f_y(x, y))$
`fchamp(f,t0,x,y)`	trace le champ de vecteurs $(x, y) \mapsto f(t_0, (x, y))$

8 Programmation

Logique : Scilab sait gérer les expressions logiques. Les commandes de base sont les suivantes. Noter que le signe `==` sert pour les tests alors que `=` sert pour donner des valeurs aux variables.

`%T`	`%F`	les valeurs booléennes "vrai" et "faux"
`&`	`| ~`	et, ou, non
`==`	`<>`	égal et différent
`<`	`>`	strictement inférieur et supérieur
`<=`	`>=`	inférieur et supérieur ou égal

Voici quelques exemples d'expressions booléennes.

```
-->%T & 1==2 , %T | 1==2, 1<>1, ~1>=0
 ans  =

  F
 ans  =

  T
 ans  =

  F
 ans  =

  F
```

Tests et boucles : Pour programmer en Scilab, on utilise trois commandes principales. La première est la condition du type `if` condition 1 `then` commandes 1 `elseif` condition 2 `then` commandes 2 `else` commandes 3 `end;`. Si condition 1 est vraie alors on effectue la liste de commandes 1, sinon, si condition 2 est vraie alors on fait la liste de commandes 2 et si les deux conditions sont fausses, on fait la liste de commandes 3. On peut mettre autant de `elseif` que voulu ou ne pas en mettre. Voici par exemple la fonction de Heaviside :

```
-->deff('y=H(x)','if x<0 then y=0; else y=1; end;')

-->H(-1), H(0), H(1)
 ans  =

    0.
 ans  =

    1.
 ans  =

    1.
```

Il y a deux sortes de boucles en Scilab. La boucle `while` condition commandes `end;` effectue la liste de commandes tant que la condition est satisfaite. ATTENTION : veillez à ce que la boucle se termine un jour. La boucle `for` variable `=`vecteur commandes `end;` effectue plusieurs fois la liste de commandes en prenant pour valeur de la variable la n-ième coordonnée du vecteur. En général, le vecteur que parcourt la variable est du type `1:n`, mais des vecteurs plus compliqués peuvent être utilisés. Voici comme illustration deux façons de calculer la somme des entiers de 1 à n.

```
-->deff('y=S1(n)','y=0; for i=1:n y=y+i; end;' )

-->deff('y=S2(n)','y=0; i=0; while i<=n y=y+i; i=i+1; end;' )

-->S1(100), S2(100), sum(1:100)
 ans  =

    5050.
 ans  =

    5050.
 ans  =

    5050.
```

Il existe deux façons de sortir d'une boucle. La commande `break` sort de la boucle en cours. La commande `return` sort du programme en cours (mais le calcul peut se poursuivre s'il s'agit d'un programme appelé par un autre programme).

`if then elseif elseif ... else end;`	des commandes exécutées sous conditions
`for i=1:n  end;`	exécute n fois la commande
`while i<=n  end;`	une boucle while
`break`	quitte la boucle en cours
`return`	quitte le programme en cours

Programmation : les programmes sont une liste de commandes Scilab écrite dans un fichier du type `toto.sci`. Ces programmes sont exécutés à l'aide de la commande `exec('toto.sci');` dans Scilab. NB : l'extension `.sci` n'a rien d'obligatoire mais elle permet de repérer facilement les fichiers Scilab dans son ordinateur (l'extension `.sce` est aussi standard). Les programmes peuvent s'auto-appeler et faire appel à des fonctions. On peut utiliser toutes les commandes mentionnées précédemment. Pour afficher un résultat, il suffit de ne pas mettre de point-virgule après la commande, ou bien de lancer un graphique. On peut aussi utiliser diverses commandes d'entrée et de sortie.
La commande `printf('chaine de caracteres',variables)` est très utile. Elle affiche la chaîne de caractères en y insérant les variables. Pour préciser quand les variables doivent apparaître, on incorpore dans la chaîne les symboles `%d %s %f %.2f` qui codent respectivement un entier, une chaîne de caractères, un nombre à virgule et un nombre avec deux chiffres derrière la virgule. On peut utiliser `\n` pour forcer le passage à la ligne.

```
-->a='ron)';b=%pi;printf('Pi=%.2f \n(envi%s',b,a);
Pi=3.14
(environ)
```

`printf('bonjour');`	écrit la chaîne de caractères
`printf('resultat=%f',a);`	écrit la valeur de la variable réelle a
`printf('%d%s%.2f\n',1,'=',1);`	affiche `1=1.00` et passe à la ligne
`pause();`	attend que l'utilisateur tape Entrée
`xpause(t);`	attend t microsecondes
`a=input('message');`	demande la valeur de a à l'utilisateur

On veillera à présenter un programme proprement et lisiblement (surtout s'il doit être lu par une autre personne, typiquement un examinateur). Voici un exemple type.

```
xpas=0.01;     // Je mets au debut la liste des variables
tpas=0.1;     // que je peux etre amene a changer souvent.
Tmax=10;       // Je leur mets un nom facilement identifiable.
Tpause=5*10^4;

//---------------------------------
// Je mets des separations entre les parties du programme
// pour voir plus clair.


deff('y=f(x,t)','y=(cos(2*%pi*x+t)).^2');


//Je mets ensuite les fonctions qui vont etre utilisees par le programme
// puis le programme en lui-meme.
//---------------------------------

x=0:xpas:1;     //Je mets ici les variables fixes dont j'ai besoin
                //mais qui ne demandent pas a etre changees souvent.

for t=0:tpas:Tmax    //L'action commence enfin.
    y=f(x,t);        //J'incremente ce qui est dans une boucle
    xbasc();
    plot2d(x,y,rect=[0,0,1,1]);
    xpause(Tpause);
end;

printf('aurevoir'); //Les resultats a afficher sont si possible a la fin.
```

Pour finir, voici une fonction calculant $n!$ en s'auto-appelant.

```
function x=fact(n)

if n<=1 then
      x=1;
    else
      x=n*fact(n-1);
end;
endfunction;
```

Fonction ou programme ? Comme on a vu, une série de commandes peut se mettre dans un fichier `toto.sci` soit telle quelle et alors on l'exécute grâce à la commande `exec('toto.sci');`, soit dans une définition de fonction `function y=f(x)` ... `endfunction`, et alors on récupère la fonction grâce à `getf('toto.sci');` puis on lance la fonction en tapant `f(x)`. Que choisir ?
Si le programme ne va être appelé que manuellement, le plus simple est la commande `exec`. On peut quand même mettre des variables (nombre de pas, précision etc.) qui sont définies au début du fichier `.sci` (afin d'y avoir accès rapidement). Cela évite d'avoir à retaper toutes les variables possibles à chaque fois qu'on relance le programme.
Si le programme est destiné à être appelé avec des variables différentes par un autre programme, alors la définition sous forme de fonction est indispensable (en particulier pour un programme s'auto-appelant).
Exemple typique : on veut tracer la précision d'une méthode d'intégration en fonction du pas de discrétisation choisi. On écrit alors une fonction qui donne le résultat de la méthode d'intégration en fonction de la précision choisie. Puis on écrit un programme, destiné à n'être lancé qu'une seule fois, qui appelle plusieurs fois la fonction précédente pour divers pas de discrétisation.

Exercice 15 : Faire un programme tirant à pile ou face et écrivant "pile" ou "face" en sortie.

Exercice 16 : Faire un programme qui demande combien on veut de tirages, puis effectue les tirages à pile ou face et donne le nombre de piles et de faces obtenus.

Exercice 17 : Faire une fonction calculant le coefficient binômial C_n^k en utilisant les formules

$$C_n^0 = C_n^n = 1 \quad C_n^k = C_{n-1}^{k-1} + C_{n-1}^k \ .$$