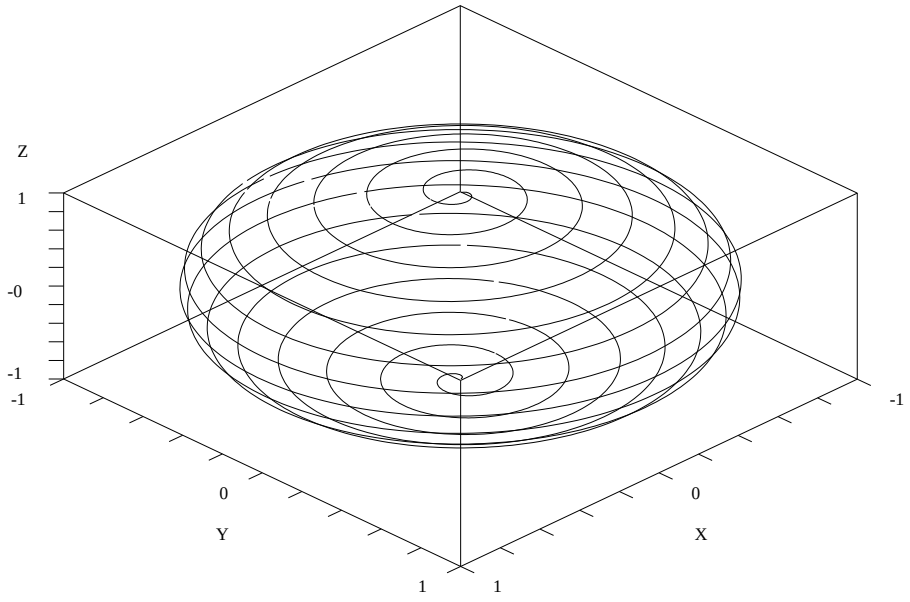
Cahiers de Mathématiques Appliquées

Démarrer en Scilab

B. Ycart

UFR Mathématiques et Informatique
Université René Descartes, Paris



Démarrer en Scilab

B. Ycart

UFR Mathématiques et Informatique
Université René Descartes, Paris
ycart@math-info.univ-paris5.fr

Scilab (contraction de *Scientific Laboratory*) est un logiciel libre, développé à l'INRIA Rocquencourt. Il est téléchargeable gratuitement à partir de :

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

C'est un environnement de calcul numérique qui permet d'effectuer rapidement toutes les résolutions et représentations graphiques couramment rencontrées en mathématiques appliquées. L'utilisation d'un tel environnement est désormais inséparable de l'activité du mathématicien : c'est un intermédiaire incontournable entre la calculatrice et un langage compilé comme C.

Il ne faut pas attendre de ce cours de démarrage une documentation complète de Scilab pas plus qu'un manuel de programmation ou un cours d'utilisation avancée. Le but de ce qui suit est d'aider le débutant en introduisant quelques unes des commandes les plus courantes. Il est conseillé de lire ce document après avoir lancé Scilab, en exécutant les commandes proposées une par une pour en observer l'effet. Les exemples ont été préparés à partir de la version 2.5 pour Linux. Compte tenu de leur caractère élémentaire, ils devraient fonctionner sur tous supports et toute version du logiciel à partir de la version 2.4.

Outre l'aide en ligne de Scilab, qui comporte de nombreux exemples, une documentation au format postscript est fournie. On trouve sur le réseau des supports de cours analogues à celui-ci, dont plusieurs sont de niveau sensiblement plus avancé.

B. Pinçon : Introduction à Scilab, *Université de Nancy, 1996.*
http://www.iecn.u-nancy.fr/~pincon/scilab/scilab.html

L.E. van Dijk et C.L. Spiel : Scilab bag of tricks, *Hammersmith Consulting, 2000.*
http ://www.hammersmith-consulting.com/scilab/sci-bot/sci-bot.html

Table des matières

1 Vecteurs et matrices

1.1 A savoir pour commencer

Scilab est basé sur le principe que tout calcul, programmation ou tracé graphique peut se faire à partir de matrices rectangulaires. En Scilab, tout est matrice : les scalaires sont des matrices 1×1, les vecteurs lignes des matrices $1 \times n$, les vecteurs colonnes des matrices $n \times 1$.

Pour démarrer, et pour une utilisation "légère", vous rentrerez des commandes ligne par ligne. Un "retour-chariot" exécute la ligne, sauf dans deux cas :

- si la ligne se termine par deux points, la séquence de commandes se prolonge sur la ligne suivante,
- si la commande définit une matrice, les lignes de cette matrice peuvent être séparées par des retours-chariots. Ceci sert essentiellement à importer de grandes matrices depuis des fichiers.

Dans une ligne de commande, tout ce qui suit `//` est ignoré, ce qui est utile pour les commentaires. Les commandes que nous proposons sur des lignes successives sont supposées être séparées par des retours-chariots.

```
A=[1,2,3;4,5,6;7,8,9]  // definit une matrice 3X3
A=[1,2,3;4,            // message d'erreur
A=[1,2,3;4,..          // attend la suite de la commande
5,6;7,8,9]             // la meme matrice est definie
A=[1,2,3;              // premiere ligne
4,5,6;                 // deuxieme ligne
7,8,9]                 // fin de la matrice
```

Ajouter un point virgule en fin de ligne supprime l'affichage du résultat (le calcul est quand même effectué). Ceci évite les longs défilements à l'écran, et s'avère vite indispensable.

```
x=ones(1,100);         // rien n'apparait
x                      // le vecteur x a bien ete defini
```

Il est fréquent que des commandes doivent être répétées avec des modifications mineures. Il est inutile de tout taper : il suffit d'appuyer sur la touche ↑ pour rappeler les commandes précédentes. On peut alors les modifier, et les exécuter à nouveau par un retour-chariot.

Dans les noms de variables, les majuscules sont distinctes des minuscules. Les résultats sont affectés par défaut à la variable `ans` ("answer"), qui contient donc le résultat du dernier calcul *non affecté*. Toutes les variables d'une session sont globales et conservées en mémoire. Des erreurs proviennent souvent de confusions avec des noms de variables déjà affectés. Il faut penser à ne pas toujours utiliser les mêmes noms, ou à libérer les variables par `clear`. Les variables courantes sont accessibles par `who` et `whos`.

```
a=[1,2]; A=[1,2;3,4];  // affecte a et A
1+1                    // affecte ans
who                    // toutes les variables
```

```
whos()                  // les details techniques
clear a
who                     // a disparait
clear
who                     // a, A et ans disparaissent
```

L'aide en ligne est appelée par `help`. La commande `apropos` permet de retrouver les rubriques d'aide quand on ignore le nom exact d'une fonction.

```
help help
help apropos
apropos matrix          // rubriques dont le titre contient "matrix"
help matrix             // aide de la fonction "matrix"
```

1.2 Construire des matrices

On peut saisir manuellement des (petites) matrices. Les coefficients d'une même ligne sont séparés par des blancs ou des virgules (préférable). les lignes sont séparées par des points-virgules. La transposée est notée par une apostrophe. Elle permet en particulier de changer un vecteur ligne en un vecteur colonne.

```
x=[1,2,3]
x'
A=[1,2,3;4,5,6;7,8,9]
A'
```

On a souvent besoin de connaître les dimensions d'une matrice (par exemple pour vérifier si un vecteur est une ligne ou une colonne). On utilise pour cela la fonction `size`.

La fonction size	
`size(A)`	nombre de lignes et de colonnes
`size(A,"r")`	nombre de lignes
`size(A,"c")`	nombre de colonnes
`size(A,"*")`	nombre total d'éléments

```
help size
A=[1,2,3;4,5,6]
size(A)
size(A')
size(A,"r")
size(A,"c")
size(A,"*")
```

Les commandes d'itération permettent de construire des vecteurs de nombres séparés par des pas positifs ou négatifs. La syntaxe de base est `[deb:pas:fin]`, qui retourne un vecteur ligne de valeurs allant de `deb` à `fin` par valeurs séparées par des multiples de `pas`. Les crochets sont facultatifs. Par défaut, `pas` vaut 1. Selon que `fin-deb` est ou non un multiple entier de `pas`, le vecteur se terminera ou non par `fin`. Si l'on souhaite un

vecteur commençant par `deb` et se terminant par `fin` avec `n` coordonnées régulièrement espacées, il est préférable d'utiliser `linspace(deb,fin,n)`.

```
v=0:10
v=0.10
v=[0:10]
v=[0;10]
v=[0,10]     // attention aux confusions
v=[0:0.1:1]
size(v)
v=[0:0.1:0.99]
size(v)
v=[0:0.15:1]
v=[1:-0.15:0]
v=linspace(0,1,10)
v=linspace(0,1,11)
```

L'élément de la `i`-ième ligne, `j`-ième colonne de la matrice `A` est `A(i,j)`. Si `v` et `w` sont deux vecteurs d'entiers, `A(v,w)` désigne la sous-matrice extraite de `A` en conservant les éléments dont l'indice de ligne est dans `v` et l'indice de colonne dans `w`. `A(v,:)` (respectivement : `A(:,w)`) désigne la sous-matrice formée des lignes (resp. : des colonnes) indicées par les coordonnées de `v` (resp. : `w`). Les coordonnées sont numérotées à partir de 1. Le dernier indice peut être désigné par `$`.

```
A=[1,2,3;4,5,6]
A(2,2)
A(:,2)
A(:,[1,3])
A(1,$)
A($,2)
```

On peut modifier un élément, une ligne, une colonne, pourvu que le résultat reste une matrice. Pour supprimer des éléments, on les affecte à la matrice vide : `[]`.

```
A=[1,2,3;4,5,6]
A(1,3)=30
A(:,[1,3])=[10,20]        // erreur
A(:,[1,3])=[10,30;40,60]
A(:,[1,3])=0
A(:,2)=[]
```

Il suffit d'une coordonnée pour repérer les éléments d'un vecteur ligne ou colonne.

```
v=[1:6]
v(3)
v(4)=40
v([2:4])=[]
w=[1:6]'
w([2,4,6])=0
```

Faire appel à un élément dont les coordonnées dépassent celles de la matrice provoque un message d'erreur. Cependant on peut étendre une matrice en affectant de nouvelles coordonnées, au-delà de sa taille. Les coordonnées intermédiaires sont nulles par défaut.

```
A=[1,2,3;4,5,6]
A(0,0)          // erreur
A(3,2)          // erreur
A(3,2)=1
v=[1:6]
v(8)            // erreur
v(8) = 1
```

Vecteurs	
`x:y`	nombres de `x` à `y` par pas de 1
`x:p:y`	nombres de `x` à `y` par pas de `p`
`linspace(x,y,n)`	`n` nombres entre `x` et `y`
`v(i)`	`i`-ième coordonnée de `v`
`v($)`	dernière coordonnée de `v`
`v(i1:i2)`	coordonnées `i1` à `i2` de `v`
`v(i1:i2)=[]`	supprimer les coordonnées `i1` à `i2` de `v`

Matrices	
`A(i,j)`	coefficient d'ordre `i,j` de `A`
`A(i1:i2,:)`	lignes `i1` à `i2` de `A`
`A($,:)`	dernière ligne de `A`
`A(i1:i2,:)=[]`	supprimer les lignes `i1` à `i2` de `A`
`A(:,j1:j2)`	colonnes `j1` à `j2` de `A`
`A(:,$)`	dernière colonne de `A`
`A(:,j1:j2)=[]`	supprimer les colonnes `j1` à `j2` de `A`
`diag(A)`	coefficients diagonaux de `A`

Si `A`, `B`, `C`, `D` sont 4 matrices, les commandes `[A,B]`, `[A;B]`, `[A,B;C,D]` retourneront des matrices construites par blocs, pourvu que les dimensions coïncident. En particulier, si `v` et `w` sont deux vecteurs lignes, `[v,w]` les concatènera, `[v;w]` les empilera.

```
A=[1,2,3;4,5,6]
M=[A,A]
M=[A;A]
M=[A,A']                    // erreur
M=[A,[10;20];[7,8,9],30]
M=[A,[10,20];[7,8,9],30]  // erreur
```

Des fonctions prédéfinies permettent de construire certaines matrices particulières.

Matrices particulières	
`zeros(m,n)`	matrice nulle de taille `m,n`
`ones(m,n)`	matrice de taille `m,n` dont les coefficients valent 1
`eye(m,n)`	matrice identité de taille min(`m,n`), complétée par des zéros
`rand(m,n)`	matrice de taille `m,n` à coefficients aléatoires uniformes sur $[0,1]$
`diag(v)`	matrice diagonale dont la diagonale est le vecteur `v`
`triu(A)`	annule les coefficients au-dessous de la diagonale
`tril(A)`	annule les coefficients au-dessus de la diagonale
`toeplitz`	matrices à diagonales constantes
`testmatrix`	carrés magiques et autres

```
d=[1:6]
D=diag(d)
A=[1,2;3;4,5,6]
[A,zeros(2,3);rand(2,2),ones(2,4)]
help testmatrix
M=testmatrix("magi",3)
help toeplitz
M=toeplitz([1:4],[1:5])
triu(M)
tril(M)
```

Les commandes `zeros`, `ones`, `eye`, `rand` retournent des matrices dont la taille peut être spécifiée soit par un nombre de lignes et de colonnes, soit par une autre matrice. Si un seul nombre est donné comme argument, ce nombre est compris comme une matrice de taille 1×1.

```
A=ones(5,5)       // 5 lignes et 5 colonnes
rand(A)           // idem
eye(A)            // idem
eye(5)            // 1 ligne et 1 colonne
ones(5)           // idem
rand(size(A))     // 1 ligne et 2 colonnes
```

On dispose de deux moyens pour réordonner des coefficients en une nouvelle matrice. La commande `A(:)` concatène toutes les colonnes de `A` en un seul vecteur colonne (on peut jouer avec la transposée pour concaténer les vecteurs lignes). La fonction `matrix` crée une matrice de dimensions spécifiées, pourvu qu'elles soient compatibles avec l'entrée. On peut même créer avec `matrix` des tableaux à plus de deux entrées (hypermatrices), ce qui ne sert que rarement.

```
v=[1:6]
help matrix
A=matrix(v,2,2)    // erreur
A=matrix(v,3,3)    // erreur
A=matrix(v,2,3)
A=matrix(v,3,2)'
```

```
w=A(:)
A=A'; w=A(:)'
H=matrix([1:24],[2,3,4])
```

Remarquons enfin qu'un peu d'algèbre permet souvent de construire des matrices particulières à peu de frais, en utilisant le produit, noté `*`, et la transposée. Le produit de Kronecker, `kron` ou `.*.`, peut également être utile.

```
A=ones(4,1)*[1:5]
A=[1:4]'*ones(1,5)
B=[1,2;3,4]
kron(B,B)
kron(B,eye(B))
kron(eye(B),B)
kron(B,ones(2,3))
kron(ones(2,3),B)
```

1.3 Opérations

Les opérations numériques s'effectuent en suivant les ordres de priorité classiques (puissance avant multiplication, et multiplication avant addition). Pour éviter les doutes il est toujours prudent de mettre des parenthèses.

```
2+3*4
(2+3)*4
2^3*4
2^(3*4)
2^3^4
(2^3)^4
```

Toutes les opérations sont matricielles. Tenter une opération entre matrices de tailles non compatibles retournera en général un message d'erreur, sauf si une des matrices est un scalaire. Dans ce cas, l'opération (addition, multiplication, puissance) s'appliquera terme à terme.

```
A=[1,2,3;4,5,6]
A+ones(1,3)     // erreur
A+ones(A)
A+10
A*10
A*ones(A)       // erreur
A*ones(A')
A'*ones(A)
```

Il n'y a pas d'ambiguïté sur l'addition, mais il est important de pouvoir appliquer des multiplications terme à terme sans qu'elles soient interprétées comme des produits matriciels. Pour cela, le signe (`*` ou `^`) doit être précédé d'un point (`.*` ou `.^`). Bien entendu les dimensions doivent être les mêmes pour qu'une opération terme à terme soit possible.

```
A=[1,2,3;4,5,6]
A*A              // erreur
A*A'
A.*A'            // erreur
A.*A
A=[1,2;3,4]
A^3
A.^3
```

La division est un cas particulier dangereux. Par défaut, `A/B` calcule une solution d'un système linéaire. Si `v` est un vecteur ligne, `1/v` retourne un vecteur colonne `w` tel que `v*w=1`. Contrairement à ce qui est écrit dans l'aide en ligne, `1./v`, compris comme `(1.0)/v`, retourne la même chose. Il y a plusieurs solutions à ce problème, dont la plus simple consiste à parenthéser : `(1)./v`.

```
v=[1,2,3]
w=1/v
v*w
1./v
(1)./v
ones(v)./v
v.^(-1)
v=v'
1/v
1./v
```

Opérations matricielles	
`+ -`	addition, soustraction
`* ^`	multiplication, puissance (matricielles)
`.* .^`	multiplication, puissance terme à terme
`A\b`	solution de `A*x=b`
`b/A`	solution de `x*A=b`
`./`	division terme à terme

1.4 Types de données

Certaines constantes sont prédéfinies, et leur valeur ne peut être modifiée.

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

Scilab est un langage faiblement typé. Les variables ne sont pas déclarées, et la même lettre peut désigner un réel, un entier, un polynôme, ou une fonction. Les types existent bien sûr, et peuvent être obtenus par la fonction `type`. Par défaut, les nombres sont traités comme des réels en double précision. Les calculs ne sont donc pas "exacts". La précision machine est donnée par la constante `%eps`, de l'ordre de 10^{-16}. Cela ne signifie pas que les nombres inférieurs à `%eps` ou supérieurs à `1/%eps` sont inaccessibles.

```
help type
%eps
1/%eps
1/%inf
1/0.00000000000000000000000000000000000000003
1/3000000000000000000000000000000000000000000
A=[1,2,3;4,5,6;7,8,8]
A/A
A*(1/A)
```

L'affichage de vecteurs avec de très grandes coordonnées réserve quelques surprises.

```
v=2^2^[1:5]
v=2^2^[1:6]       // etonnant non ?
v(1:5)            // le calcul etait correct
v=2^2^[1:10]
log(v)./log(2)
2^[1:10]
```

A part les réels, les entrées d'une matrice peuvent être des complexes, des booléens, des polynômes ou des chaînes de caractères.

Les booléens sont `%t` (true) et `%f` (false) pour les saisies, `T` et `F` pour les affichages. L'égalité logique est notée `==` pour la distinguer de l'affectation.

```
a=2==3
b=2<3
c=%t
a|(b&c)
```

Opérateurs logiques

`==`	$=$	`~`	non
`<`	$<$	`>`	$>$
`<=`	$\leq$	`>=`	$\geq$
`&`	et	`|`	ou

Les matrices booléennes sont le résultat d'opérations logiques matricielles (toujours terme à terme). Par exemple, si `A` et `B` sont deux matrices (réelles) de même taille, `A<B` est la matrice des booléens `A(i,j)<B(i,j)`. Les opérateurs `and` et `or` peuvent s'appliquer à l'ensemble des éléments d'une matrice booléenne, à ses lignes ou à ses colonnes selon l'option choisie. La commande `find(v)` retourne la liste des indices des coordonnées "vraies" du vecteur booléen `v`. Si `v` est une matrice, ses colonnes sont

préalablement concaténées. Si `A` est une matrice et `B` est une matrice de booléens, la commande `A(B)` extrait de `A` les coordonnées correspondant aux indices pour lesquels `B` est à vrai. Cette propriété, fort utile en programmation, a pour corollaire quelques comportements curieux : `ones(B)` retournera une matrice de même taille que `B`, mais `rand(B)` retourne un message d'erreur. La fonction `bool2s` transforme des booléens en 0 ou 1.

```
x=rand(2,10)
b=x<0.5
bool2s(b)
and(b)
and(b,"c")
and(b,"r")
or(b,"r")
b1=b(1,:); b2=b(2,:);
b1 & b2
b1 | b2
find(b1)
y=[1:10]
y(b1)
A=[1,2,3;4,5,6;7,8,9]
x=[%t,%f,%t]
A(x,x)
A(~x,x)
B=rand(3,3)<0.5
A(B)
```

Les complexes sont définis à l'aide de la constante `%i` ($\sqrt{-1}$) ou bien en affectant `sqrt(-1)` à une nouvelle variable. A noter que par défaut `A'` est la transposée *conjuguée* de `A`. La transposée non conjuguée est `A.'`. Les fonctions classiques appliquées à un complexe donnent toujours un résultat unique, même si mathématiquement elles ne sont pas définies de façon unique (racine carrée, logarithme, puissances).

```
A=[1,2;3,4]+%i*[0,1;2,3]
real(A)
imag(A)
conj(A)
A'
A.'
abs(A)
phasemag(A)
i=sqrt(-1)
%i*i
A*i
%e^(%i*%pi)
x=log(%i)
```

```
exp(x)
x=%i^(1/3)
x^3
```

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

Les polynômes et les fractions rationnelles constituent un type de données particulier. On peut construire un polynôme comme résultat d'opérations sur d'autres polynômes. Il existe par défaut un polynôme élémentaire, `%s`. On peut construire un polynôme d'une variable quelconque à l'aide de la fonction `poly`, en spécifiant soit ses racines (par défaut), soit ses coefficients. Les fractions rationnelles sont des quotients de polynômes. Par défaut, Scilab effectue automatiquement les simplifications qu'il reconnaît.

```
apropos poly
help poly
v=[1,2,3]
p1=poly(v,"x")
roots(p1)
p2=poly(v,"x","c")
coeff(p2)
p1+p2
p3=1+2*%s-3*%s^2
p1+p3                    // erreur : variables differentes
p4=(%s-1)*(%s-2)*(%s-3)
p3/p4
```

Polynômes	
`poly(v,"x")`	polynôme dont les racines sont les éléments de `v`
`poly(v,"x","c")`	polynôme dont les coefficients sont les éléments de `v`
`inv_coeff(v)`	idem
`coeff(P)`	coefficients du polynôme `P`
`roots(P)`	racines du polynôme `P`
`factors`	facteurs irréductibles réels d'un polynôme

Les chaînes de caractères, encadrées par des doubles côtes (`"..."`), permettent de définir des expressions mathématiques, interprétables ensuite comme des commandes à exécuter ou des fonctions à définir. Elles servent aussi d'intermédiaire pour des échanges de données avec des fichiers. On peut donc transformer et formater des nombres en chaînes de caractères (voir `apropos string`).

```
expression=["x+y","x-y"]     // matrice de 2 chaines de caracteres
length(expression)           // longueurs des 2 chaines
size(expression)             // taille de la matrice
part(expression,2)           // deuxieme caractere de chaque chaine
expression(1)+expression(2)  // concatenation des deux chaines
x=1; y=2;
evstr(expression)
x=1; instruction="y=(x+1)^2"
execstr(instruction)
y
deff("p=plus(x,y)","p=x+y")
plus(1,2)
plus(2,3)
```

Chaînes de caractères	
`evstr`	évaluer une expression
`deff`	définir une fonction
`execstr`	exécuter une instruction
`length`	longueur
`part`	extraire
`+`	concaténer
`string`	transformer en chaîne

1.5 Fonctions

Scilab propose beaucoup de fonctions dans des domaines très variés. On peut en obtenir la description par `help`. On peut retrouver une fonction avec `apropos`, qui retourne les rubriques d'aide dont le titre contient une chaîne de caractères donnée. De très nombreux exemples sont disponibles en démonstration (bouton `demo` du menu `file` ou ligne de commande).

```
help sin
apropos sin
exec("SCI/demos/alldems.dem");
```

Les fonctions numériques s'appliquent en général à chaque terme d'un vecteur ou d'une matrice. Il ne faut pas oublier que la multiplication, la puissance, la division doivent être précédées par un point pour s'appliquer terme à terme.

```
x=1:10
y=sin(x)
y=x*sin(x)       // erreur
y=x.*sin(x)
y=1./x.*sin(x)   // erreur
y=(1)./x.*sin(x)
y=sin(x)./x
```

Fonctions mathématiques		
sqrt	exp	log
sin	cos	tan
asin	acos	atan
round	floor	ceil

Les fonctions vectorielles s'appliquent à l'ensemble d'un vecteur ou d'une matrice, et retournent un scalaire. Pour appliquer une telle fonction colonne par colonne ou ligne par ligne, il faut rajouter l'option `"r"` ou `"c"`. Il n'est pas toujours évident de décider laquelle des deux options choisir. Il faut se souvenir qu'avec l'option `"r"`, la fonction retournera un vecteur ligne (row), et donc s'appliquera aux colonnes. Par exemple, `sum(A,"r")` retourne un vecteur ligne, qui est formé des sommes des coefficients dans chaque colonne.

Fonctions vectorielles	
max	maximum
min	minimum
sort	tri par ordre décroissant
sortup	tri par ordre croissant
gsort	tri, ordres particuliers
sum	somme
prod	produit
cumsum	sommes cumulées
cumprod	produits cumulés
mean	moyenne
median	médiane
st_deviation	écart-type

```
A=[1,2,3;4,5,6]
sum(A)
sum(A,"r")
sum(A,"c")
cumsum(A,"r")
cumsum(A,"c")
cumsum(A)
x=rand(1,5)
mean(x)
st_deviation(x)
median(x)
sort(x)
sortup(x)
gsort(x,"c","i")
```

L'utilisation de Scilab consiste en général à étendre le langage par de nouvelles fonctions, définies par des séquences d'instructions. Nous avons vu l'utilisation de `deff`, nous verrons plus loin la syntaxe des fichiers de fonctions. Il est important de choisir des

noms différents pour ces nouvelles fonctions, sans quoi les définitions se superposeront, y compris à celles des fonctions prédéfinies.

```
deff("y=sin(x)","y=2*x")       // message d'avertissement
sin(2)
```

Les nouvelles fonctions sont traitées comme des variables, à la différence des fonctions prédéfinies (primitives) : une fonction définie par `deff` peut être utilisée comme argument dans une nouvelle fonction mais une primitive ne peut pas.

```
deff("y=pente_secante(f,x,y)","y=(f(y)-f(x))/(y-x)")
x=%pi; y=%pi+0.01;
pente_secante(cos,x,y)         // erreur
deff("y=f(x)","y=cos(x)")
pente_secante(f,x,y)
z=[0:0.01:%pi];
fplot2d(z,cos)                 // erreur
fplot2d(z,f)
```

Quand on définit une nouvelle fonction numérique, on a toujours intérêt à faire en sorte qu'elle puisse s'appliquer correctement à une matrice, ce qui impose de veiller aux multiplications terme à terme. On peut aussi utiliser aussi `feval`, qui distribue l'évaluation d'une fonction sur l'ensemble des éléments d'un vecteur.

```
deff("y=f(x)","y=x*sin(x)")
f(1)
f([1:5])                       // erreur
help feval
feval([1:5],f)
deff("y=g(x)","y=x.*sin(x)")
g(1)
g([1:5])
```

Certaines fonctions peuvent retourner plus d'un argument. Par exemple les fonctions de tri `sort`, `sortup` et `gsort` retournent par défaut le vecteur trié, mais peuvent aussi donner la permutation des coordonnées qui a été effectuée.

```
v=rand(1,5)
sort(v)
[vtrie,perm]=sort(v)
```

2 Graphiques

Le principe général des représentations graphiques est de se ramener à des calculs sur des matrices ou des vecteurs. Ainsi la représentation d'une fonction de $\mathbb{R}$ dans $\mathbb{R}$ commencera par la création d'un vecteur d'abscisses, en général régulièrement espacées, auxquelles on applique la fonction pour créer le vecteur des ordonnées. Pour la

représentation d'une surface, il faudra créer la matrice des valeurs de la fonction sur une grille rectangulaire dans $\mathbb{R}^2$.

Il est impossible de décrire ici l'ensemble des fonctions graphiques et leurs multiples options. Certaines de ces options, comme la numérotation des couleurs, sont globales et peuvent être fixées par `xset` (voir `help xset`). La commande `xset()` ouvre un panneau de contrôle. Les démonstrations donnent une bonne idée des possibilités graphiques de Scilab. On obtient en général un exemple d'utilisation d'une fonction graphique en appelant cette fonction à vide. L'interface permet de zoomer sur les images 2D et d'effectuer des rotations sur les images 3D à l'aide de la souris.

Par défaut, les graphiques successifs sont superposés sur la même fenêtre. On efface la fenêtre courante par `xbasc()`. On ouvre la fenêtre numéro `i` par `xset("window",i)`.

```
help Graphics
plot2d1()
xbasc()
histplot()
xbasc()
plot3d()
xbasc()
hist3d()
xbasc()
param3d()
```

2.1 Représenter des fonctions

Le plus simple pour commencer est de tracer le graphe d'une fonction de $\mathbb{R}$ dans $\mathbb{R}$ à l'aide de `plot`. On crée pour cela un vecteur `x` d'abscisses, et on prend l'image de ce vecteur par la fonction pour créer un vecteur `y` d'ordonnées. La commande `plot(x,y)` représente les points de coordonnées `(x(i),y(i))` en les joignant par des traits noirs (par défaut), ou selon un autre style, si le style de base a été changé. La qualité de la représentation dépend donc du nombre de points. On pourra essayer l'exemple suivant en changeant le style de graphique par `xset()`. Notez que contrairement aux autres primitives graphiques, `plot` efface la fenêtre à chaque appel.

```
x=linspace(0,3*%pi,10); y=x.*sin(x);
plot(x,y)
x=linspace(0,3*%pi,100); y=x.*sin(x);
plot(x,y)
```

On obtient le même résultat par `fplot2d`, mais il faut pour cela prédéfinir la fonction à représenter. Les tracés successifs se superposent.

```
deff("y=f(x)","y=x.*sin(x)")
x=linspace(0,3*%pi,10);
fplot2d(x,f)
x=linspace(0,3*%pi,100);
```

`fplot2d(x,f)`

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

```
plot2d(abscisses,ordonnees,style,cadre,legendes,bornes,graduation)
```

Après les deux premiers, les arguments sont facultatifs, mais si l'un d'entre eux est précisé, ceux qui le précèdent devront l'être aussi. La signification des arguments est la suivante.

- `abscisses, ordonnees` : ce sont nécessairement des matrices de mêmes dimensions. Si ce sont des vecteurs (une seule courbe à tracer), ils peuvent être ligne ou colonne. Si plusieurs courbes doivent être tracées, elles doivent correspondre à autant de colonnes. Par défaut les points seront reliés par des segments. A chaque courbe correspond une couleur de la palette (il y en a 32).

```
x=linspace(0,3*%pi,30);
y=x.*sin(x);
plot2d(x,y)
xbasc()
y2=2*y;
plot2d([x,x],[y,y2])   // incorrect : deux courbes concatenees
xbasc()
plot2d([x;x],[y;y2])   // incorrect : trace 30 segments
xbasc()
plot2d([x;x]',[y;y2]') // correct
xbasc()
X=x'*ones(1,20);
Y=y'*[1:20];
plot2d(X,Y)
```

- `style` : c'est un vecteur ligne dont la dimension est le nombre de courbes à tracer (nombre de colonnes des matrices `abscisses` et `ordonnees`). Les coordonnées sont positives ou négatives. Si le style est positif, les points sont joints par des segments. Si le style est nul, les points sont affichés comme des pixels noirs. Si le style est négatif, des marques de formes particulières sont affichées.

```
x=linspace(0,3*%pi,30); X=x'*ones(1,10);
y=x.*sin(x); Y=y'*[1:10];
couleurs=matrix([2;5]*ones(1,5),1,10)
xbasc()
plot2d(X,Y,couleurs)
marques=-[0:9]
xbasc()
plot2d(X,Y,marques)
```

- `cadre` : ce paramètre est une chaîne de caractères formée de trois chiffres, dont :
  – le premier code la présence de légendes (`0` ou `1`),

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

Par défaut, l'argument `cadre` vaut `"021"` (pas de légendes, échelles de représentation calculées automatiquement, axes tracés). Si l'on superpose deux graphiques avec cette option par défaut, les échelles ne seront pas les mêmes. La solution consiste à tracer tous les graphiques à partir du second sur une même fenêtre avec l'option `"000"` (pas de légende, utiliser les échelles précédentes, ne pas retracer les axes).

```
x=linspace(0,3*%pi,30);
y=x.*sin(x);
xbasc()
plot2d(x,y)
y2=2*y;
plot2d(x,y2)         // incorrect : les echelles sont differentes
xbasc();
plot2d(x,y)
plot2d(x,y2,"000")   // erreur
plot2d(x,y2,1,"000")
xbasc();
plot2d(x,y2)
plot2d(x,y,1,"000")
```

- `legendes` : c'est une chaîne de caractères contenant les différentes légendes, séparées par `@`.

```
x=linspace(0,3*%pi,30); X=x'*ones(1,5);
y=x.*sin(x); Y=y'*[1:5];
styles=[-2:2]
legendes="x sin(x)@2 x sin(x)@3 x sin(x)@4 x sin(x)@5 x sin(x)"
xbasc()
plot2d(X,Y,styles,"121",legendes)
```

- `bornes` : c'est le rectangle de représentation, décrit par les deux coordonnées du coin inférieur gauche, suivies des deux coordonnées du coin inférieur droit : `[xmin,ymin,xmax,ymax]`.

```
x=linspace(0,3*%pi,30);
y=x.*sin(x);
xbasc()
plot2d(x,y,1,"011"," ",[0,-10,10,10])
```

- `graduations` : ce vecteur de quatre entiers permet de préciser la fréquence des graduations et sous-graduations en abscisse et ordonnée. Par exemple, avec `[2,10,4,5]`, l'intervalle des abscisses sera divisé en 10, chacun des 10 sous-intervalles étant subdivisé en 2. Pour les ordonnées il y aura 5 sous-intervalles, chacun subdivisé en 4.

```
x=linspace(0,3*%pi,30);
y=x.*sin(x);
xbasc()
```

```
plot2d(x,y,1,"011"," ",[0,-10,10,10],[2,10,4,5])
```

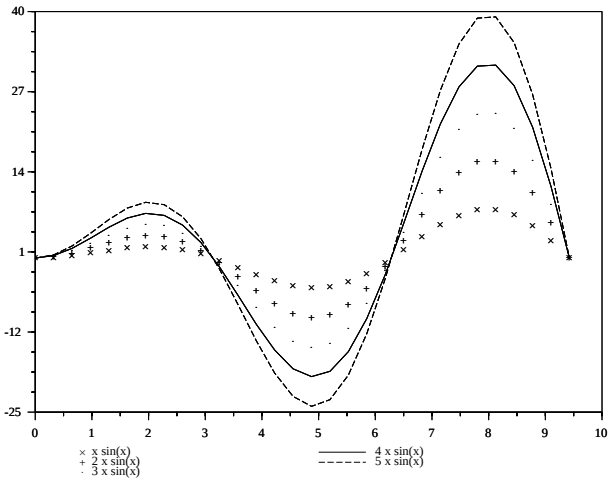


FIG. 1 – Représentation de plusieurs courbes.

Graphes de fonctions	
`plotframe`	rectangle de représentation
`plot`	points joints par des segments
`plot2d`	plusieurs courbes avec styles différents
`plot2d1`	idem, avec plus d'options
`plot2d2`	représentation en escalier
`plot2d3`	barres verticales
`plot2d4`	flèches
`fplot2d`	représenter des fonctions

Les échelles de représentation, choisies automatiquement, ne sont pas les mêmes en abscisse et en ordonnée. On peut corriger ceci à l'aide de `isoview` ou `square`.

2.2 Graphiques composés

Si une même représentation graphique comporte plusieurs tracés à des échelles différentes, il vaut mieux spécifier d'abord le rectangle de représentation et les échelles des axes par `plotframe`, pour superposer ensuite les différents tracés.

```
xbasc();
xset("font",2,4);
plotframe([-4,-1,4,1],[2,10,5,10],[%f,%f],["Titre","Axe x","Axe y"]);
x=linspace(-%pi,%pi,50); y=sin(x);
plot2d(x,y,1,"000");              // trace une courbe
x=linspace(-%pi/2,%pi/2,5); y=sin(x);
xset("mark",1,4);
```

```
plot2d(x,y,-3,"000");            // affiche 5 marques
x=linspace(-%pi,%pi,20); y=sin(x)/2;
xset("use color",0);
xset("pattern",13);
xfpoly(x,y);                     // surface grisee
xset("default");
```

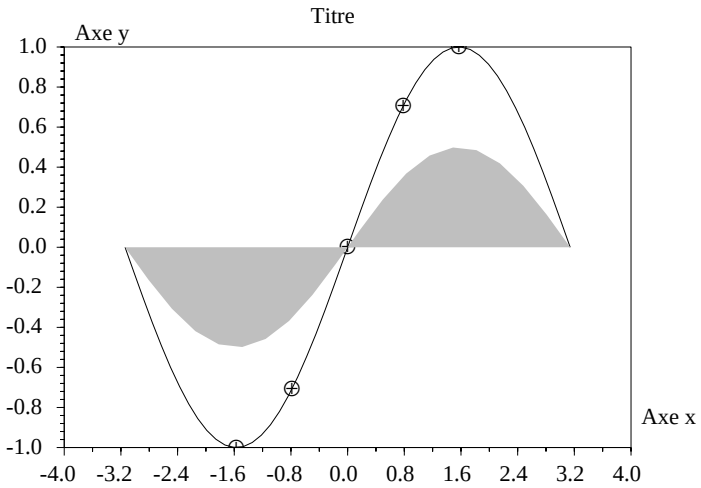


FIG. 2 – Figure composée.

Il est fréquent qu'un graphique contienne non seulement une ou plusieurs représentations de fonctions, mais aussi des chaînes de caractères, des rectangles, ellipses ou autres ajouts graphiques. Les coordonnées de ces ajouts sont relatives à la fenêtre courante. Là aussi, on aura intérêt à spécifier le cadre au préalable par `plotframe`.

```
xbasc()
plotframe([-1,0,2,4],[10,3,5,4],[%f,%f],["Parabole","x","f(x)"])
x=linspace(-1,2,100); y=x.*x;
plot2d(x,y,2,"000")              // represente la courbe
plot2d([1,1,-1],[0,1,1],3,"000") // trace deux segments
help xstring
xstring(1.1,0.1,"abscisse")      // chaine de caracteres
xstring(-0.9,1.1,"ordonnee")     // autre chaine
help xarc
xarc(-0.5,1,1,1,0,360*64)        // trace un cercle
```

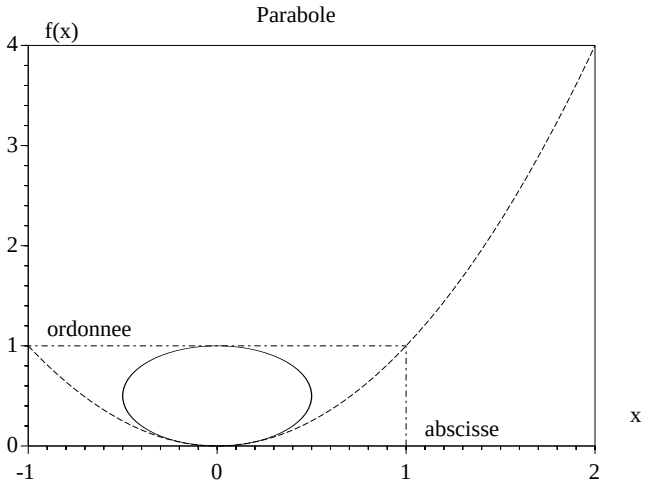


FIG. 3 – Figure composée.

Ajouts sur graphique	
`xarc`	arc d'ellipse
`xfarc`	arc d'ellipse plein
`xarrows`	flèches
`xnumb`	nombres
`xpoly`	polygone
`xfpoly`	polygone plein
`xrpoly`	polygone régulier
`xrect`	rectangle
`xfrect`	rectangle plein
`xstring`	chaîne de caractères (à partir d'un point)
`xstringb`	chaîne de caractères (dans un rectangle)
`xtitle`	titre du graphique et des axes

Des fonctions prédéfinies permettent d'effectuer des représentations planes particulières, comme des histogrammes, des projections de surfaces par courbes de niveau ou niveaux de gris, ou des champs de vecteurs. Les exemples qui suivent concernent la fonction de $\mathbb{R}^2$ dans $\mathbb{R}$ qui à (x, y) associe $\sin(xy)$ (voir figures 4, 5 et 6).

```
xbasc()
//
// Courbes de niveau
//
x=linspace(-%pi,%pi,50);      // vecteur d'abscisses
y=x;                          // vecteur d'ordonnees
z=sin(x'*y);                  // matrice des valeurs de la fonction
help contour2d
xbasc()
```

```
contour2d(x,y,z,4)            // trace 3 courbes de niveau
//
// Surface par niveaux de couleurs
//
xbasc()
grayplot(x,y,z)               // pas vraiment gray le plot
xbasc()
R=[0:255]/256; G=R; B=R;
RGB=[R;G;B]';                 // nouvelle matrice de couleurs
xset("colormap",RGB);
grayplot(x,y,z)               // niveaux de gris
xset("default")               // reinitialise les parametres graphiques
//
// Champ de vecteurs tangents
//
x=linspace(-%pi,%pi,10);      // vecteur d'abscisses
y=x;                          // vecteur d'ordonnees
fx=cos(x'*y)*diag(y);         // matrice des abscisses de vecteurs
fy=diag(x)*cos(x'*y);         // matrice des ordonnees de vecteurs
champ(x,y,fx,fy)              // champ des vecteurs
```

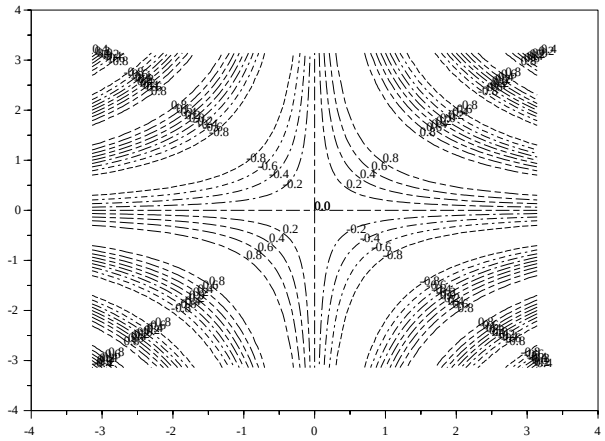


FIG. 4 – Représentation par courbes de niveau.

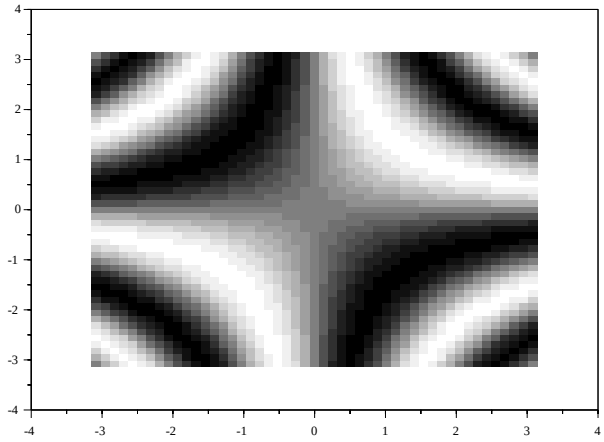


FIG. 5 – Représentation par niveaux de gris.

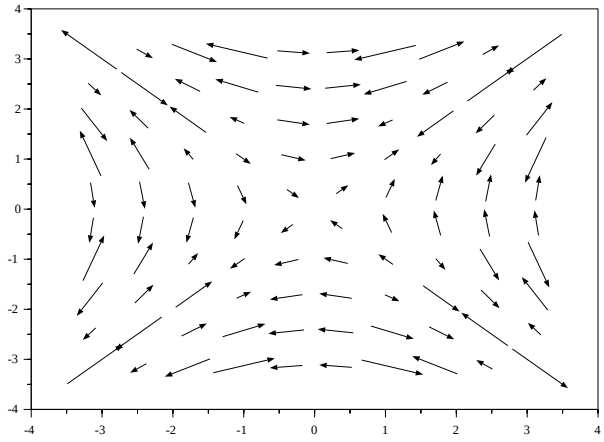


FIG. 6 – Représentation d'un champ de vecteurs.

Représentations planes particulières	
`histplot`	histogramme
`champ`	champ de vecteurs
`fchamp`	idem, définition par une fonction
`grayplot`	surface par rectangles de couleurs
`fgrayplot`	idem, définition par une fonction
`contour2d`	courbes de niveaux projetées
`fcontour2d`	idem, définition par une fonction

2.3 Dimension 3

Le tracé d'une courbe en dimension 3 se fait par la fonction `param3d`, selon les mêmes principes qu'en dimension 2.

```
xbasc()
t=linspace(0,2*%pi,50);
x=sin(t); y=sin(2*t); z=sin(3*t);
param3d(x,y,z)                // courbe de Lissajous
xbasc()
t=linspace(-%pi/2,%pi/2,1000);
x=cos(t*50).*cos(t);
y=sin(t*50).*cos(t);
z=sin(t);
param3d(x,y,z)                // helice spherique
```

Pour représenter une famille de courbes en dimension 3, il faut utiliser `param3d1`. Les arguments sont trois matrices de coordonnées pour lesquelles les différentes courbes sont en colonne.

```
xbasc()
t=linspace(0,2*%pi,100);
a=linspace(-%pi,%pi,10);
X=cos(t')*cos(a);             // matrice des abscisses
Y=sin(t')*cos(a);             // matrice des ordonnees
Z=ones(t')*sin(a);            // matrice des cotes
param3d1(X,Y,Z)               // paralleles d'une sphere
```

La représentation des surfaces se fait par `plot3d` ou `plot3d1`. Nous reprenons comme exemple la fonction de $\mathbb{R}^2$ dans $\mathbb{R}$ qui à (x, y) associe $\sin(xy)$ (figure 7).

```
x=linspace(-%pi,%pi,50);      // vecteur d'abscisses
y=x;                          // vecteur d'ordonnees
z=sin(x'*y);                  // matrice des valeurs de la fonction
help plot3d
xbasc()
plot3d(x,y,z)                 // representation monochrome
plot3d1(x,y,z)                // representation coloree
xbasc()
R=[0:255]/256; G=R; B=0.5*ones(R);
RGB=[R;G;B]';                 // nouvelle matrice de couleurs
xset("colormap",RGB);
plot3d1(x,y,z)                // les couleurs dependent de z
xset("default")               // reinitialise les parametres graphiques
```

Pour représenter une surface définie par deux paramètres, il faut la définir comme une fonction, puis utiliser `eval3dp` qui prend comme argument cette fonction et deux vecteurs de paramètres, et retourne les arguments nécessaires pour la représentation

par `plot3d`. Voici par exemple la représentation d'une sphère. Pour obtenir des couleurs variables, il faut parfois changer le sens d'un des deux vecteurs de paramètres.

```
deff("[x,y,z]=sphere(u,v)",.. // definition de la fonction
["x=cos(u).*cos(v);..         // abscisses
y=sin(u).*cos(v);..           // ordonnees
z=sin(v)"])                   // cotes

u = linspace(-%pi,%pi,50);
v = linspace(-%pi/2,%pi/2,25);// parametres
[x,y,z] = eval3dp(sphere,u,v);// calcul de la surface
plot3d1(x,y,z);               // representation monochrome

u = linspace(%pi,-%pi,50);    // changement de sens
[x,y,z] = eval3dp(sphere,u,v);// nouveau calcul
xbasc()
plot3d1(x,y,z)                // les couleurs dependent de z
```

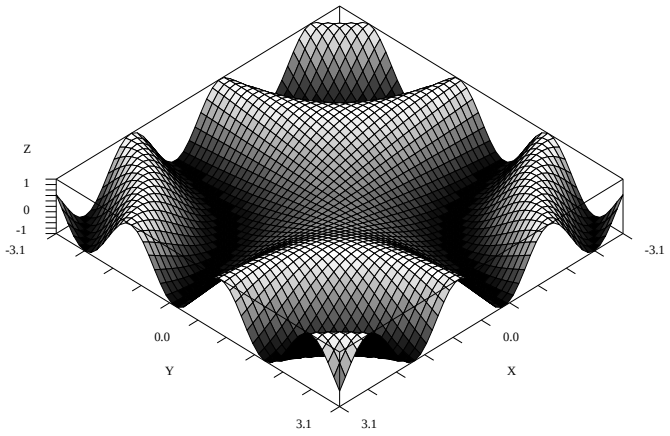


FIG. 7 – Représentation par `plot3d1`.

Dimension 3	
`param3d`	courbes paramétriques
`param3d1`	plusieurs courbes ou points
`plot3d`	surface en dimension 3
`fplot3d`	idem, définition par une fonction
`plot3d1`	surface par niveaux de couleurs
`fplot3d1`	idem, définition par une fonction
`eval3dp`	surface paramétrée
`hist3d`	histogrammes

2.4 Exporter des graphiques

Chaque fenêtre graphique propose une commande "export" qui permet de sauvegarder la figure au format postscript (pour l'insérer dans un texte latex par exemple) ou au format xfig, ce qui permet de la modifier ou de la compléter. Les courbes en couleur sont (parfois) traduites en noir et blanc par des styles de pointillés.

Ces sauvegardes d'images réservent quelques surprises mineures. Sauvegardez aux deux formats, postscript et xfig, la figure produite par les lignes de commande suivantes, et comparez avec la fenêtre Scilab.

```
xset("font",2,3);
xbasc();
plotframe([0,0,10,10],[0,1,0,1],[%f,%f],["essai de marques","",""]);
for i=1:9,
  for j=1:9,
     xset("mark",-i,j);
     plot2d(i,j,-i,"000");
  end;
end;
```

Les trois figures sont légèrement différentes. Les formes de certaines marques (triangles, trèfle) changent dans la version postscript. Les bords en haut et à droite apparaissent en pointillés dans la version xfig. La figure 8 représente la sauvegarde postcript (à gauche) et la version xfig (à droite).

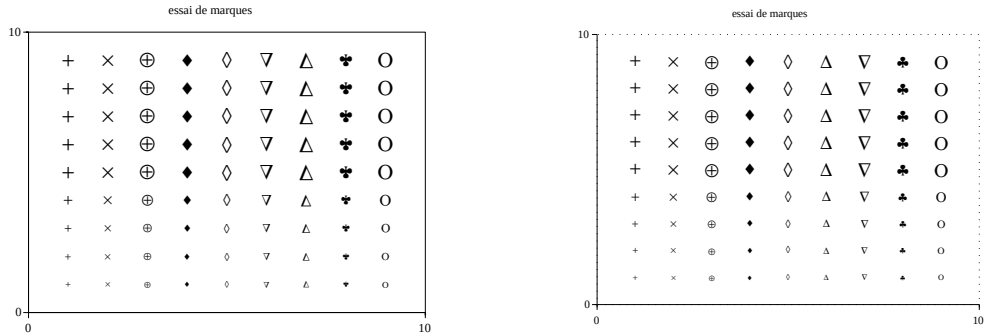


FIG. 8 – Essai de marques : export au format postscript (à gauche) et au format xfig (à droite).

3 Calcul numérique

3.1 Algèbre linéaire

Tout traitement mathématique est vu par Scilab comme une manipulation de matrices. Pour l'algèbre linéaire, les fonctionnalités sont donc particulièrement dévelop-

pées. Dans une utilisation un peu avancée, on aura intérêt à tirer parti de la structure de matrice creuse (`sparse`) que nous n'aborderons pas.

Il est important de garder à l'esprit que tous les calculs numériques dépendent de la représentation des nombres réels en machine, et que donc les résultats "exacts" n'existent pas. La détection de singularités dépend d'une précision prédéfinie et n'est donc pas infaillible.

```
rank([1,1;1,1])
rank([1.00000000000001,1;1,1])
rank([1.000000000000001,1;1,1])
inv([1,1;1,1])
A=[1,2,3;4,5,6;7,8,9]
det(A)
rank(A)
inv(A)     // le calcul est effectue
A*inv(A)   // ce n'est pas l'identite
```

Si `A` et `B` sont deux matrices, alors `A\B` retourne une matrice `X` telle que `A*X=B`, et `B/A` une matrice `X` telle que `X*A=B` pourvu que les dimensions de `A` et `B` soient compatibles. Si `A` n'est pas une matrice carrée inversible, ces commandes retourneront un résultat, qu'il y ait une infinité de solutions ou qu'il y en ait aucune.

```
A=[1,2,3;4,5,6;7,8,9]
b=[1;1;1]
x=A\b       // le systeme a une infinite de solutions
A*x
b=[1,1,1]
x=b/A       // le systeme a une infinite de solutions
x*A
b=[1;1;2]
x=A\b       // le systeme n'a pas de solution
A*x
b=[1,1,2]
x=b/A       // le systeme n'a pas de solution
x*A
```

Pour résoudre un système linéaire quand on craint une singularité, il vaut mieux utiliser `linsolve`, qui détecte les impossibilités et peut retourner une base du noyau de la matrice en cas de solutions multiples. Attention, `linsolve(A,b)` résout `A*x=-b`.

```
A=[1,2,3;4,5,6;7,8,9]
b=[1;1;1]
[x,k]=linsolve(A,b)
A*x
A*k
b=[1;1;2]
[x,k]=linsolve(A,b)  // erreur
```

La commande `[D,U]=bdiag(A)` retourne (en théorie) une matrice diagonale `D` et une matrice inversible `U` telles que `U*D*inv(U)=A`.

```
A=diag([1,2,3]); P=rand(3,3); A=P*A*inv(P)
spec(A)
[D,U]=bdiag(A)       // matrice diagonalisable
inv(U)*A*U
U*D*inv(U)-A
A=[1,0,0;0,1,1;0,0,1]; P=rand(3,3); A=P*A*inv(P)
spec(A)
[D,U]=bdiag(A)       // matrice non diagonalisable
inv(U)*A*U
U*D*inv(U)-A
```

Opérations matricielles	
`A'`	transposée de `A`
`rank`	rang
`inv`	inverse
`expm`	exponentielle matricielle
`det`	déterminant
`trace`	trace
`poly(A,"x")`	polynôme caractéristique de `A`
`spec`	valeurs propres de `A`
`bdiag`	diagonalisation
`svd`	décomposition en valeurs singulières
`A\b`	solution de `A*x=b`
`b/A`	solution de `x*A=b`
`linsolve(A,b)`	solution de `A*x=-b`

3.2 Intégration

Des fonctions d'intégration numérique sont disponibles pour les fonctions réelles et complexes, à une, deux et trois variables. Les fonctions `integrate`, `intg`, `int2d`, `int3d`, `intc` et `intl` prennent en entrée une fonction externe, ou définie par une chaîne de caractères. Les fonctions `integ`, `inttrap` et `intsplin` prennent en entrée des vecteurs d'abscisses et d'ordonnées.

Calculs d'intégrales	
`integrate`	fonction définie par une chaîne de caractères
`intg`	fonction externe
`integ`	vecteurs d'abscisses et d'ordonnées
`inttrap`	méthode des trapèzes
`intsplin`	approximation par splines
`int2d`	fonction de deux variables
`int3d`	fonction de trois variables
`intc`	fonction complexe le long d'un segment
`intl`	fonction complexe le long d'un arc de cercle

Dans l'exemple ci-dessous, on calcule avec les différentes fonctions d'intégration disponibles, la valeur de :

$$\int_{-10}^{0} e^x \, dx \;=\; 1 - e^{-10} \;\simeq\; 0.9999546 \, .$$

```
x = [-10:0.1:0];
y=exp(x);
1-1/%e^10
inttrap(x,y)
intsplin(x,y)
integ(y,x)
integrate("exp(x)","x",-10,0)
deff("y=f(x)","y=exp(x)")
intg(-10,0,f)
```

Les algorithmes de calcul numérique des transformées classiques sont disponibles. Le tableau ci-dessous en donne quelques-unes, voir `apropos transform` pour les autres.

Transformées	
`dft`	transformée de Fourier discrète
`fft`	transformée de Fourier rapide
`convol`	produit de convolution
`flt`	transformée de Legendre rapide
`dmt`	transformée de Mellin discrète
`cwt`	transformée en ondelette continue

3.3 Résolutions et optimisation

La fonction `fsolve` résout numériquement un système d'équations, mis sous la forme $f(x) = 0$. Comme toutes les résolutions numériques, celle-ci part d'une valeur initiale x_0, et itère une suite censée converger vers une solution. Le résultat dépend évidemment de la valeur initiale.

```
deff("y=f(x)","y=sin(%pi*x)")
fsolve(0.2,f)
fsolve(0.4,f)
fsolve(0.45,f)
fsolve(0.5,f)
fsolve([0.45:0.01:0.5],f)
help fsolve
[x,v,info]=fsolve(0.5,f)
[x,v,info]=fsolve([0.45:0.01:0.5],f)
```

Résolutions	
`fsolve`	systèmes d'équations
`roots`	racines d'un polynôme
`factors`	facteurs irréductibles réels d'un polynôme
`linsolve`	systèmes linéaires

La fonction `optim` recherche un minimum (local) d'une fonction dont on connaît le gradient. La définition de la fonction en entrée est un peu particulière. Le choix de trois algorithmes différents est offert en option.

```
help optim
deff("[y,yprim,ind]=f(x,ind)","y=sin(%pi*x),yprim=%pi*cos(%pi*x)")
[x,v]=optim(f,0.2)
[v,xopt]=optim(f,0.50000000000000001)
[v,xopt]=optim(f,0.5000000000000001)
```

Optimisation	
`optim`	optimisation
`linpro`	programmation linéaire
`quapro`	programmation quadratique

3.4 Equations différentielles

La fonction `ode` est en fait un environnement qui donne accès à la plupart des méthodes numériques classiques pour la résolution des équations différentielles ordinaires (voir `help ode`).

A titre d'exemple nous commençons par le problème de Cauchy en dimension 1 suivant :

$$\begin{cases} y'(t) &= y(t)\cos(t)\,, \\ y(0) &= 1\,. \end{cases}$$

La solution explicite est $y(t) = \exp(\sin(t))$. La résolution numérique par `ode` est très stable.

```
deff("yprim=f(t,y)","yprim=y*cos(t)")
t0=0; y0=1; t=[0:0.01:10];
sol=ode(y0,t0,t,f);
max(abs(sol-exp(sin(t))))     // l'erreur est tres faible
t0=0; y0=1; t=[0:0.1:100];
sol=ode(y0,t0,t,f);
max(abs(sol-exp(sin(t))))     // l'erreur reste tres faible
```

La situation n'est pas toujours aussi favorable. Considérons par exemple $y(t) = \exp(t)$, solution du problème de Cauchy suivant.

$$\begin{cases} y'(t) &= y(t)\,, \\ y(0) &= 1\,. \end{cases}$$

```
deff("yprim=f(t,y)","yprim=y")
t0=0; y0=1; t=[0:0.01:10];
sol=ode(y0,t0,t,f);
max(abs(sol-exp(t)))          // l'erreur est raisonnable
t0=0; y0=1; t=[0:0.1:100];
sol=ode(y0,t0,t,f);
max(abs(sol-exp(t)))          // l'erreur est enorme
```

Voici en dimension 2 la résolution d'un système différentiel de Lotka-Volterra, avec superposition de la trajectoire calculée et du champ de vecteurs défini par le système (figure 9).

```
//
// definition du systeme
//
deff("yprim=f(t,y)",..
    ["yprim1=y(1)-y(1)*y(2)";..
     "yprim2=-2*y(2)+2*y(1)*y(2)";..
     "yprim=[yprim1;yprim2]"])
//
// champ de vecteurs
//
xmin=0; xmax=3; ymin=0; ymax=3;
fx=xmin:0.3:xmax; fy=ymin:0.3:ymax;
xbasc()
fchamp(f,1,fx,fy)
//
//resolution du systeme
//
t0=0; tmax=5; pas=0.1
t=t0:pas:tmax;
y0=[2;2];
sol=ode(y0,t0,t,f);
//
// trace de la trajectoire
//
plot2d(sol(1,:),sol(2,:),5,"000")
```

4 Programmation

4.1 Types de fichiers

Scilab travaille à partir d'un répertoire de base, qui est donné par la commande `pwd`. C'est là qu'il va chercher par défaut les fichiers à charger ou à exécuter. On peut le changer par la commande `chdir`. A défaut, il faut saisir le chemin d'accès complet

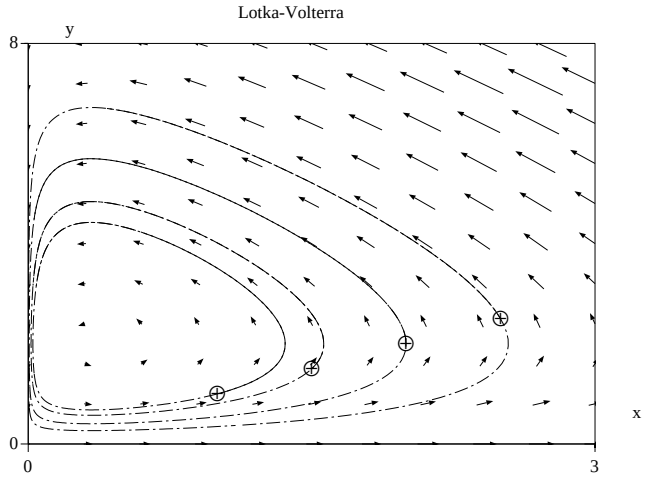


FIG. 9 – Résolution d'un système de Lotka-Volterra par `ode`, représentation de plusieurs solutions et du champ de vecteurs.

du fichier que l'on souhaite charger ou sauvegarder. Le plus facile est d'utiliser le menu de l'interface.

Dès que les calculs à effectuer requièrent plus de quelques lignes de commande, on a intérêt à saisir ces lignes dans un fichier exécutable externe. Dans l'interface de Scilab, les seules commandes qui apparaîtront seront les exécutions ou les chargements répétés de fichiers externes. Il est conseillé de maintenir ouvertes deux fenêtres : la fenêtre Scilab, et une fenêtre d'édition (Emacs sous Linux, WordPad sous Windows, par exemple). Scilab distingue trois sortes de fichiers.

1. *Les fichiers de sauvegarde.* Ce sont des fichiers binaires, créés par la commande `save` et rappelés par `load`. Ceci permet de reprendre un calcul en conservant les valeurs déjà affectées. On peut aussi enregistrer des variables dans un fichier texte par `write` et les rappeler par `read`.
2. *Les fichiers de commandes.* Ce sont des fichiers texte. Ils contiennent des suites d'instructions Scilab, qui sont exécutées successivement par `exec`. Enregistrez dans le répertoire courant les trois lignes suivantes sous le nom `losange.sce`. Attention, la dernière ligne du fichier doit obligatoirement se terminer par un retour-chariot pour être prise en compte.

   ```
   x=[0,-1,0,1;-1,0,1,0]
   y=[-1,0,1,0;0,1,0,-1]
   plot(x,y)
   ```

   La commande `exec("losange.sce")` affichera `x`, puis `y`, puis tracera un losange.

   On peut écrire une matrice dans un fichier texte à exécuter (par exemple pour importer des données issues d'un tableur). Dans ce cas, les lignes du fichier, si elles correspondent aux lignes de la matrice, ne doivent pas se terminer par deux

points. Par exemple le fichier `saisie.txt` peut contenir les trois lignes (terminées par un retour-chariot) :

```
A=[1,2,3;
4,5,6;
7,8,9];
```

La commande `exec("saisie.txt")` affectera la matrice `A`.

3. *Les fichiers de fonctions.* Comme les fichiers de commandes, ce sont des fichiers texte. Ils contiennent la définition d'une ou plusieurs fonctions. La définition d'une fonction commence obligatoirement par une ligne qui déclare le nom de la fonction, les variables d'entrée `x1,x2,...,xm` et le vecteur des variables de sortie `[y1,y2,...,yn]`.

   ```
   function [y1,y2,...,yn] = nom_de_la_fonction(x1,x2,...,xm)
   ```

   Là encore, il ne faut pas oublier de terminer la dernière ligne par un retour-chariot. Enregistrez par exemple dans le fichier `cloche.sci` les lignes suivantes.

   ```
   function d = cloche(x)
   // CLOCHE densite de la loi normale N(0,1)
   d = (1/sqrt(2*%pi))*exp(-x.^2/2);
   ```

   Si ce fichier est placé dans le répertoire courant, `getf("cloche.sci")` charge et compile la nouvelle fonction.

   ```
   getf("cloche.sci")
   x=[-3:0.1:3]; y=cloche(x);
   plot(x,y)
   fplot2d(x,cloche)
   intg(-5,1.96,cloche)
   ```

Les fichiers les plus couramment utilisés sont les fichiers de commandes et les fichiers de fonctions. Les extensions `.sce` pour les fichiers de commandes et `.sci` pour les fichiers de fonctions sont de tradition, mais ce n'est pas une obligation. Un fichier de commandes peut réaliser les mêmes tâches qu'une fonction et réciproquement. Pour une utilisation courante ou de mise au point, les fichiers de commandes permettent de suivre le contenu de toutes les variables. Pour une programmation plus avancée, il est préférable de définir des fonctions, car leurs variables internes restent locales. Un même fichier `.sci` peut contenir plusieurs fonctions. Les fonctions du langage sont regroupées dans des librairies qui contiennent leur code Scilab (fichiers texte `.sci`), et leur code compilé (fichiers `.bin`). On peut transformer un ensemble de fichiers de fonctions en librairie, en sauvant les versions compilées et en rajoutant les fichiers d'aide.

4.2 Style de programmation

La philosophie de Scilab est celle d'un langage fonctionnel. Au lieu de créer un logiciel avec programme principal et procédures, on étend le langage par les fonctions

dont on a besoin. Le rôle du programme principal est joué par un fichier de commandes contenant essentiellement des appels de fonctions.

Certaines erreurs difficiles à trouver proviennent de confusions entre noms de variables ou de fonctions. Scilab garde en mémoire tous les noms introduits tant qu'ils n'ont pas été libérés par `clear`. Il est donc prudent de donner des noms assez explicites aux variables. Les variables introduites dans la session ou dans les fichiers de commandes sont globales. Par défaut, toutes les variables introduites à l'intérieur d'une fonction sont locales. C'est une des raisons pour lesquelles on a intérêt à définir de nouvelles fonctions plutôt que d'utiliser des fichiers de commande exécutables.

Pour comparer l'efficacité des algorithmes, on dispose de `timer` qui permet de compter le temps CPU écoulé.

```
A=rand(200,200);
b=rand(200,1);
timer(); x=A\b; timer()      // resout le systeme lineaire
timer(); x=inv(A)*b; timer() // inverse la matrice : plus lent
```

Scilab propose les commandes des langages de programmation classiques.

Commandes principales				
Pour	`for`	`x=vecteur,`	`instruction;`	`end`
Tant que	`while`	`booleen,`	`instruction;`	`end`
Si	`if`	`booleen then,`	`instruction;`	`end`
Sinon	`else`		`instruction;`	`end`
Sinon si	`elseif`	`booleen then,`	`instruction;`	`end`
Selon	`select x`	`case 1 ...`	`then ...`	`end`

La boucle `for` peut aussi s'appliquer à une matrice. Dans :

```
for x=A, instruction, end
```

l'instruction sera exécutée pour `x` prenant successivement comme valeurs les colonnes de la matrice `A`.

Scilab est un outil de calcul plus que de développement. Pour un problème compliqué, on aura tendance à utiliser Scilab pour réaliser des maquettes de logiciels et tester des algorithmes, quitte à lancer ensuite les gros calculs dans un langage compilé comme C. Cela ne dispense pas de chercher à optimiser la programmation en Scilab, en utilisant au mieux la logique du calcul matriciel. Voici un exemple illustrant cette logique. Si $v = (v_i)_{i=1\ldots n}$ et $w = (w_j)_{j=1\ldots m}$ sont deux vecteurs, on souhaite définir la matrice $A = (a_{i,j})$, où $a_{i,j} = v(i)^{w(j)}$. Il y a plusieurs solutions. Dans les commandes qui suivent, `v` est un vecteur colonne et `w` est un vecteur ligne.

```
for i=1:n, for j=1:m, A(i,j)=v(i)^w(j); end; end // a eviter
A=v^w(1); for j=2:m, A=[A,v^w(j)]; end           // mieux
A=v(1)^w; for i=2:n, A=[A;v(i)^w]; end           // equivalent
A=(v*ones(w)).^(ones(v)*w)                       // preferable
```

Si on doit appeler plusieurs fois ce calcul, on aura intérêt à en faire une fonction.

Scilab offre toutes les facilités pour programmer correctement : protection des saisies, utilitaires de débuggage...

Protection et débuggage	
`disp`	affichage de variables
`type`	type des variables
`typeof`	idem
`argn`	nombre de variables d'entrée
`break`	sortie de boucle
`pause`	attente clavier
`return`	sortie de fonction
`resume`	idem
`error`	message d'erreur
`warning`	message d'avertissement

Nous donnerons plusieurs exemples de fonctions dans le chapitre suivant. Elles ne sont pas toujours optimisées. Nous avons privilégié la clarté de la programmation sur la rapidité d'exécution.

5 Statistiques en Scilab

Scilab propose dans sa version de base un éventail complet et bien structuré de fonctions de simulation et de traitement statistique. On y trouve des générateurs pseudo-aléatoires, des représentations graphiques, des calculs de moments et des calculs sur les lois de probabilités usuelles. L'utilisateur peut s'il le souhaite, compléter cet éventail par quelques fonctions simples pour en faire un outil de traitement statistique tout à fait performant.

5.1 Lois discrètes

Nous verrons à la section suivante que le générateur `grand` permet la simulation des lois discrètes usuelles. Voici une fonction de génération d'une loi discrète quelconque, par la méthode d'inversion.

```
function e = ech_dist(m,n,x,d)
//      ech_dist(m,n,x,d) retourne une matrice de taille mXn dont les
//      coefficients sont des realisations independantes de la loi
//      sur x specifiee par le vecteur d, normalise a 1.

d = d/sum(d);                    // normaliser par la somme
loi=cumsum(d);                   // calculer la fonction de repartition

for i=1:m,
     for j=1:n,
          k=1;
          r=rand(1,1);           // appel de random
          while r>loi(k),        // simulation par inversion
               k=k+1,
```

```
          end;
          e(i,j)=x(k);
     end;
end
```

La fonction suivante nous servira à calculer les fréquences des différentes valeurs apparaissant dans un échantillon.

```
function [f,v] = freq_emp(ech)
//      freq_emp(ech) Calcule les frequences empiriques des valeurs
//      differentes de ech. Retourne un vecteur des valeurs differentes
//      de ech et un vecteur des frequences correspondantes.

taille=size(ech,"*");              // taille de l'echantillon

v=[ech(1)];                        // valeurs differentes
for k = 2:taille,                  // parcourir l'echantillon
     if ech(k)~=v then,            // la valeur v(k) est nouvelle
          v = [v,ech(k)];          // la rajouter
     end;
end;
v = gsort(v,"c","i");              // trier les valeurs trouvees
nbval=size(v,"*");                 // nombre de valeurs differentes

effectifs = [];                    // effectifs des valeurs
for k = 1:nbval,                   // parcourir les valeurs
     e = size(find(ech==v(k)),"*");// calculer l'effectif de v(k)
     effectifs = [effectifs,e];    // le rajouter
end;

f=effectifs/sum(effectifs);        // calculer les frequences
```

Nous supposons que les deux fonctions `ech_dist` et `freq_emp` ont été placées dans le fichier `lois_discretes.sci`. Voici deux exemples d'utilisation.

```
getf("lois_discretes.sci")
x = ech_dist(1,100,["a","b","c"],[0.5,0.3,0.2])
x = ech_dist(1,1000,["a","b","c"],[0.5,0.3,0.2]);
[f,v] = freq_emp(x)
plot2d3("gnn",[1:3]',f',5,"111","frequences",[0,0,4,1])
x = ech_dist(1,1000,[1:10],ones(1,10));
[f,v] = freq_emp(x)
xbasc()
plot2d3("gnn",v',f',5,"111","frequences",[0,0,11,0.5])
```

5.2 Générateurs pseudo-aléatoires

Les deux fonctions de génération aléatoire sont `rand` et `grand`. Le plus simple est `rand`. Il retourne des réalisations de variables i.i.d, de loi uniforme sur $[0,1]$ ou de loi normale $\mathcal{N}(0,1)$, selon l'option.

```
help rand
// Loi uniforme
unif = rand(1,1000);
histplot(10,unif);
moyennes=cumsum(unif)./[1:1000];
xbasc()
plot(moyennes)
// Loi normale
gauss = rand(1,1000,"normal");
xbasc()
histplot(10,gauss)
// Pile ou face
getf("lois_discretes.sci")
pileface = 2*bool2s(rand(1,1000)<0.5)-1;
[f,v] = freq_emp(pileface)
gains = cumsum(pileface);
plot(gains)
// Loi binomiale
binomiale=sum(rand(4,1000)<0.5,"r");
[f,v] = freq_emp(binomiale)
// Loi exponentielle
expo=-log(unif);
mean(expo)
median(expo)
st_deviation(expo)
xbasc()
histplot(10,expo)
// Loi geometrique
geom = ceil(expo);
[f,v] = freq_emp(geom)
xbasc()
plot2d3("gnn",v',f',5,"111","Geometrique",[0,0,8,max(f)])
```

Le générateur `grand` est très complet et nous ne présentons qu'une partie de ses possibilités (voir `help grand`). Dans les appels de `grand` pour une loi unidimensionnelle, les deux premiers paramètres `m` et `n` spécifient le nombre de lignes et de colonnes de l'échantillon à engendrer. Comme pour `rand`, ces deux nombres peuvent être remplacés par un vecteur ou une matrice.

```
getf("lois_discretes.sci")
// Loi binomiale
```

```
b=grand(1,1000,"bin",10,0.7);
[f,v]=freq_emp(b)
xbasc()
plot2d3("gnn",v',f',5,"111","Binomiale",[0,0,11,max(f)])
// Loi de Poisson
p=grand(b,"poi",2);
[f,v]=freq_emp(p)
xbasc()
plot2d3("gnn",v',f',5,"111","Poisson",[0,0,8,max(f)])
// Loi du khi-deux
c1=grand(1,1000,"chi",3);
xbasc()
histplot(20,c1)
// Loi gamma
c2=grand(c1,"gam",3/2,1/2);
xbasc()
histplot(20,c2)
// Loi normale dans le plan
M=[0;0]; S=[1,0.8;0.8,1];
X=grand(5000,"mn",M,S);
xbasc()
square(-1,1,-1,1);
plot2d(X(1,:),X(2,:),0);
// Chaine de Markov
P=[0,0.4,0.6;0.2,0.3,0.5;0.5,0.5,0];
X=grand(10000,"markov",P,1);
[f,v]=freq_emp(X)
[x,sta]=linsolve(P'-eye(P),zeros(3,1));
sta=sta/sum(sta,"r")       // mesure stationnaire
abs(f'-sta)
```

Simulations	
`grand(m,n,"bet",A,B)`	loi bêta
`grand(m,n,"bin",N,p)`	loi binomiale
`grand(m,n,"chi",N)`	loi du khi-deux
`grand(m,n,"def")`	loi uniforme sur $[0,1]$
`grand(m,n,"exp",M)`	loi exponentielle
`grand(m,n,"f",M,N)`	loi de Fisher
`grand(m,n,"gam",A,L)`	loi gamma
`grand(m,n,"lgi")`	loi uniforme sur $\{0,\ldots,2^{31}\}$
`grand(n,"mn",M,S)`	loi normale multidimensionnelle
`grand(n,"markov",P,x0)`	chaîne de Markov
`grand(n,"mul",N,P)`	loi multinomiale
`grand(m,n,"nbn",N,p)`	loi binomiale négative
`grand(m,n,"nch",N,x0)`	loi du khi-deux décentrée
`grand(m,n,"nf",M,N,x0)`	loi de Fisher décentrée
`grand(m,n,"nor",M,S)`	loi normale unidimensionnelle
`grand(m,n,"poi",L)`	loi de Poisson
`grand(m,"prm",V)`	permutations aléatoires de `V`
`grand(m,n,"uin",A,B)`	entiers uniformes entre `A` et `B`
`grand(m,n,"unf",A,B)`	réels uniformes entre `A` et `B`

5.3 Représentations graphiques

La fonction `plot2d3` permet de représenter des diagrammes en bâtons et a déjà été utilisée. La fonction `histplot` représente des histogrammes. Son premier paramètre peut être un entier (nombre de classes), auquel cas l'histogramme est régulier, ou un vecteur donnant les bornes des classes.

```
x=rand(1,1000);
xbasc()
histplot(10,x)
xbasc()
histplot([0,0.2,0.5,0.6,0.9,1],x)
y=grand(x,"exp",1);
xbasc()
histplot(10,y)
xbasc()
histplot(-log([1:-0.1:0.01]),y)
```

Pour représenter un nuage de points dans le plan, on peut utiliser `plot2d` avec un style de représentation négatif ou nul. La fonction `hist3d` représente des histogrammes dans $I\!R^3$ mais n'effectue pas le calcul des fréquences de classes. Pour cela, on pourra utiliser la fonction `freq2d` définie ci-dessous. Elle prend en entrée deux vecteurs de bornes, `bornex` et `borney`, et une matrice `echant`, à 2 lignes et n colonnes.

```
function f = freq2d(echant,bornex,borney)
```

```
//      freq2d retourne une matrice dont les
//      coefficients sont les frequences de echant
//      relatives a bornex et borney

kx = size(bornex,"*")-1;
ky = size(borney,"*")-1;

for i=1:kx,
    for j=1:ky,
        f(i,j) = size(find(..
            echant(1,:) >  bornex(i)   &..
            echant(1,:) <= bornex(i+1) &..
            echant(2,:) >  borney(j)   &..
            echant(2,:) <= borney(j+1)),"*");
   end;
end;
f=f/sum(f);
```

Cette fonction sera enregistrée dans le fichier `freq2d.sci`. La fonction `hist3d` prend en entrée une liste formée de la matrice des fréquences, retournée par `freq2d`, et des deux vecteurs de bornes. Voici quelques exemples d'utilisation.

```
getf("freq2d.sci")
u=rand(2,2000);
plot2d(u(1,:),u(2,:),0)
bx=[0:0.2:1]; by=bx;
f=freq2d(u,bx,by);
xbasc()
hist3d(list(f,bx,by))

t1=max(rand(2,2000),"r");
t2=min(rand(2,2000),"r");
xbasc()
plot2d(t1,t2,0)
bx=[0:0.2:1]; by=bx;
f=freq2d([t1;t2],bx,by);
xbasc()
hist3d(list(f,bx,by))

M=[0;0]; S=[1,0.8;0.8,1];
v=grand(2000,"mn",M,S);
xbasc()
plot2d(v(1,:),v(2,:),0)
bx=[-3:1:3]; by=bx;
f=freq2d(v,bx,by);
xbasc()
```

```
hist3d(list(f,bx,by))
```

On peut visualiser des nuages de points à trois dimensions à l'aide de `param3d1`, et utiliser la rotation à l'aide de la souris.

```
M=[0;0;0];
S=[1,0.5,0.9 ; 0.5,1,0.3 ; 0.9,0.3,1];
v=grand(10000,"mn",M,S);
xbasc()
param3d1(v(1,:)',v(2,:)',list(v(3,:)',0))
```

5.4 Calculs de moments

Les calculs de moyennes, médianes et écart-types s'effectuent par des fonctions vectorielles (`mean`, `median` et `st_deviation`) qui s'appliquent à l'ensemble d'une matrice, à ses lignes ou à ses colonnes selon l'option. Attention, `st_deviation` calcule la racine carrée de la variance *non biaisée*.

La fonction `corr` retourne les corrélations ou covariances d'une ou deux séries chronologiques, mais ne s'applique pas directement au calcul de la matrice de covariance d'un échantillon vectoriel. Les fonctions ci-dessous calculent les matrices de covariance et de corrélation pour un échantillon de taille n de vecteurs de dimension d, donné sous forme d'une matrice à n lignes et d colonnes.

```
function c=covariance(echant)
//      retourne la matrice de covariance
//      de l'echantillon echant. L'entree est une matrice
//      a n lignes (individus) et d colonnes (variables)

moyenne = mean(echant,"r");
echcent = echant - ones(size(echant,"r"),1)*moyenne;
c=(echcent'*echcent)/size(echant,"r");


function c=correlation(echant)
//      retourne la matrice de correlation
//      de l'echantillon echant. L'entree est une matrice
//      a n lignes (individus) et d colonnes (variables)

cov=covariance(echant);
ectinv=diag((1)./sqrt(diag(cov)));
c=ectinv*cov*ectinv;
```

Supposons que ces deux fonctions soient dans le fichier `covariance.sci`.

```
getf("covariance.sci");
M=[0;0;0];
S=[1,0.5,0.9 ; 0.5,1,0.3 ; 0.9,0.3,1];
v=grand(10000,"mn",M,S);
```

```
cov_estimee = covariance(v')
cov_estimee-S
cor_estimee = correlation(v')
cor_estimee-S
```

5.5 Analyse de données

Les fonctionnalités de Scilab en algèbre linéaire facilitent évidemment les techniques d'analyse descriptive de tableaux de données comme l'analyse en composantes principales, l'analyse factorielle, l'analyse discriminante, et autres. A titre d'exemple, nous donnons ci-dessous une fonction réalisant l'ACP réduite d'un tableau de données. La fonction affiche la projection des individus et des variables sur le premier plan principal, et retourne les pourcentages d'inertie expliqués par les axes principaux. Il est facile de la compléter pour qu'elle affiche la projection sur d'autres plans, ou de la modifier pour en déduire d'autres types d'analyses.

```
function c=ACP_Reduite(echant);
//      Affiche les projections des individus et des variables
//      sur le premier plan principal de l'ACP reduite.
//      Retourne les pourcentages d'inertie expliquee.
//      L'entree est une matrice a n lignes (individus)
//      et d colonnes (variables).

//
// Calculs de moments
//
moyenne = mean(echant,"r");            // moyennes par variable
ni = size(echant,"r");                 // nombre d'individus
nv = size(echant,"c");                 // nombre de variables
echred = echant - ones(ni,1)*moyenne;  // echantillon centre
mat_cov = (echred'*echred)/ni;         // matrice de covariance
ecti = diag((1)./sqrt(diag(mat_cov))); // inverses des ecart-types
echred = echred*ecti;                  // echantillon reduit
mat_cor = ecti*mat_cov*ecti;           // matrice de correlations
//
// Calculs des projections sur les axes principaux
//
[D,U] = bdiag(mat_cor);                // diagonalisation
[c,k] = sort(diag(D));                 // tri des valeurs propres
c = round((c/sum(c,"r"))*100);         // pourcentages d'inertie
axes = U(:,k);                         // reordonner les axes propres
indiv = echred*axes;                   // nouvelles coordonnees
//
// Representation graphique plan 1-2
//
```

```
e = max(abs(indiv));                    // echelle de representation
xset("window",0); xbasc();              // preparation du graphique
xset("font",2,3);
plotframe([-e,-e,e,e],[2,10,2,10],[%f,%f],..
   ["Projection sur le plan 1-2","1er Axe","2eme Axe"]);
plot2d(indiv(:,1),indiv(:,2),0,"000"); // afficher les individus
cvar = axes([1,2],:);                   // coordonnees des variables
cvar = cvar.*.[1,0];                    // insertion de l'origine
cvar = cvar';
                                        // afficher les variables
plot2d(cvar(:,1),cvar(:,2),5*ones(1,nv),"000")
```

Voici un exemple d'utilisation (la fonction est dans le fichier `acp.sci`).

```
getf("acp.sci");
M1 = zeros(3,1);
V1 = eye(3,3);
X1 = grand(1000,"mn",M1,V1);
M2 = 3*ones(3,1);
V2 = [1,0.8,0.6;0.8,1,0.2;0.6,0.2,1];
X2 = grand(1000,"mn",M2,V2);
X = [X1';X2'];
inertie = ACP_Reduite(X)
```

5.6 Calculs sur les lois usuelles

Scilab propose des fonctions `cdf*`, à partir desquelles on retrouve la fonction de répartition, la densité et la fonction quantile des lois les plus courantes. Ces fonctions sont les suivantes.

Lois usuelles	
`cdfbet`	lois bêta
`cdfbin`	lois binomiales
`cdfchi`	lois du khi-deux
`cdfchn`	lois du khi-deux décentrées
`cdff`	lois de Fisher
`cdffnc`	lois de Fisher décentrées
`cdfgam`	lois gamma
`cdfnbn`	lois binomiales négatives
`cdfnor`	lois normales unidimensionnelles
`cdfpoi`	lois de Poisson
`cdft`	lois de Student

Les familles de lois de probabilité classiques dépendent de un, deux voire trois paramètres. Pour chaque valeur du ou des paramètres, la fonction de répartition de la loi correspondante est déterminée de façon unique. C'est une fonction qui à une valeur

réelle x associe une probabilité $p = F(x)$. L'inverse de la fonction de répartition est la fonction quantile, qui à une probabilité p associe le p-quantile x. Entrent donc en jeu :

- les valeurs des paramètres,
- la valeur du quantile x,
- la valeur de la probabilité p.

Toutes les fonctions `cdf*` sont structurées de façon analogue. Si on leur donne en entrée toutes les quantités sauf une, ainsi que l'option appropriée, la fonction retournera la quantité manquante. Les variables d'entrée étant des vecteurs, on peut effectuer plusieurs calculs simultanés sur la même loi ou sur des lois différentes.

Prenons d'abord l'exemple de la loi binomiale (fonction `cdfbin`). L'option `"PQ"` permet de calculer des valeurs de fonctions de répartition ou de leurs compléments à 1. A partir de ces valeurs, il est facile par différences de retrouver les valeurs des probabilités. Les deux exemples qui suivent concernent la loi binomiale de paramètres 10 et 0.6.

```
help cdfbin
[P,Q]=cdfbin("PQ",5,10,0.6,0.4)
c=ones(1,11);
repart=cdfbin("PQ",0:10,10*c,0.6*c,0.4*c)
xbasc()
plot2d2("gnn",[0:10]',repart',5,"111","Repartition",[0,0,11,1])
repart=[0,repart];
probas=repart(2:12)-repart(1:11)
xbasc()
plot2d3("gnn",[0:10]',probas',5,"111","Probabilites",[0,0,10,0.3])
esperance=sum(probas.*[0:10])
```

L'option `"S"` permet de calculer les valeurs de la fonction quantile (interpolée).

```
cdfbin("S",10,0.2,0.8,0.6,0.4)
p=[0.1:0.1:1];
c=ones(p);
cdfbin("S",10*c,p,1-p,0.6*c,0.4*c)
```

Les autres options `"Xn"` et `"PrOmpr"` permettent de retouver un paramètre de la loi connaissant l'autre paramètre et une valeur de la fonction de répartition.

Voici maintenant l'exemple de la loi normale (fonction `cdfnor`). Là encore l'option `"PQ"` permet de calculer la fonction de répartition. Sa différentielle discrète est une approximation de la densité.

```
help cdfnor
[P,Q] = cdfnor("PQ",1.96,0,1)
x=linspace(-3,3,100);
repart=cdfnor("PQ",x,zeros(x),ones(x));
plot(x,repart)
dens=(repart(2:100)-repart(1:99))*100/6;
x(1)=[];
plot(x,dens)
```

Pour la fonction quantile, il faut utiliser l'option "X".

```
cdfnor("X",0,1,0.975,0.025)
p=[0.01:0.01:0.99];
quant=cdfnor("X",zeros(p),ones(p),p,1-p);
plot(p,quant)
```

Les options "Mean" et "Std" permettent de calculer la moyenne ou la variance d'une loi normale dont on connaît l'autre paramètre et une valeur de la fonction de répartition.

Les fonctions cdf* suffisent pour toutes les applications de statistique inférentielle classique : elles permettent de calculer les bornes des intervalles de confiance, les régions de rejet et les p-valeurs de tous les tests usuels. Voici par exemple le calcul de l'intervalle de confiance de niveau 0.95 pour l'espérance d'une loi normale, basé sur la loi de Student.

```
n = 30
echant = grand(1,n,"nor",2,3);
moy = mean(echant)
v = mean(echant.*echant)-moy*moy
quantile = cdft("T",n-1,0.975,0.025)
amplitude = quantile*sqrt(v/(n-1))
intervalle = moy+[-1,1]*amplitude
```

Voici maintenant l'exemple du test du khi-deux. La fonction khideux ci-dessous prend en entrée une distribution empirique, une distribution théorique et une taille d'échantillon. Elle calcule la distance du khi-deux, et retourne la p-valeur du test du khi-deux, pour le cas le plus simple où aucun paramètre n'a été estimé.

```
function pv = khideux(de,dt,n)
//      retourne la p-valeur du test du khi-deux pour la
//      la distribution empirique "de" et la distribution
//      theorique "dt", et une taille d'echantillon "n".

c = size(de,"*");              // nombre de classes
d = sum(((de-dt).^2)./dt);     // distance du khi-deux
s = n*d;                       // statistique de test
[p,pv] = cdfchi("PQ",s,c-1);  // calcul de la p-valeur
```

Pour tester cette fonction (placée dans le fichier khideux.sci), nous utilisons à nouveau les fonctions ech_dist et freq_emp du fichier lois_discretes.sci.

```
getf("lois_discretes.sci");
getf("khideux.sci");
n=100; dt = [0.5,0.3,0.2]
ech = ech_dist(1,n,[1,2,3],dt);
de = freq_emp(ech)
pv = khideux(de,dt,n)
```

6 Exercices

Il y a souvent plusieurs manières d'obtenir le même résultat en Scilab. On s'efforcera de choisir les solutions les plus compactes, c'est-à-dire celles qui utilisent au mieux le langage matriciel.

Exercice 1 Ecrire (sans utiliser de boucle) les vecteurs suivants :

1. Nombres de 1 à 3 par pas de 0.1.
2. Nombres de 3 à 1 par pas de -0.1.
3. Carrés des 10 premiers entiers.
4. Nombres de la forme $(-1)^n n^2$ pour $n = 1, \ldots, 10$.
5. 10 "0" suivis de 10 "1".
6. 3 "0" suivis de 3 "1", suivis de 3 "2",..., suivis de 3 "9".
7. 1 "1" suivi de 2 "2", suivis de 3 "3",..., suivis de 9 "9".
8. "1", suivi de 1 "0", suivi de "2", suivi de 2 "0",..., suivi de "8", suivi de 8 zéros, suivi de "9".

Exercice 2 Ecrire (sans utiliser de boucle) les matrices carrées d'ordre 6 suivantes :

1. Matrice diagonale, dont la diagonale contient les entiers de 1 à 6.
2. Matrice contenant les entiers de 1 à 36, rangés par lignes.
3. Matrice dont toutes les lignes sont égales au vecteur des entiers de 1 à 6.
4. Matrice diagonale par blocs, contenant un bloc d'ordre 2 et un d'ordre 4. Les 4 coefficients du premier bloc sont égaux à 2. Le deuxième bloc contient les entiers de 1 à 16 rangés sur 4 colonnes.
5. Matrice $A = ((-1)^{i+j})$, $i, j = 1, \ldots, 6$.
6. Matrice contenant des "1" sur la diagonale, des "2" au-dessus et au-dessous, puis des "3", jusqu'aux coefficients d'ordre $(1, 6)$ et $(6, 1)$ qui valent 6.

Exercice 3 Ecrire la matrice $A = (a_{i,j})$ d'ordre 12 contenant les entiers de 1 à 144, rangés par lignes. Extraire de cette matrice les matrices B suivantes.

1. Coefficients $a_{i,j}$ pour $i = 1, \ldots, 6$ et $j = 7, \ldots, 12$.
2. Coefficients $a_{i,j}$ pour $i + j$ pair.
3. Coefficients $a_{i,j}$ pour $i, j = 1, 2, 5, 6, 9, 10$.

Exercice 4

1. Ecrire les polynômes de degré 6 suivants.
   (a) polynôme dont les racines sont les entiers de 1 à 6.
   (b) polynôme dont les racines sont 0 (racine triple), 1 (racine double) et 2 (racine simple).
   (c) polynôme $(x^2 - 1)^3$.
   (d) polynôme $x^6 - 1$.

Pour chacun de ces polynômes :

2. Ecrire la matrice compagnon A associée à ce polynôme : la matrice compagnon associée au polynôme :

$$P = x^d - a_{d-1}x^{d-1} - \cdots - a_1 x - a_0 \ ,$$

est :

$$A = \begin{pmatrix} 0 & 1 & 0 & \dots & & 0 \\ \vdots & \ddots & \ddots & \ddots & & \vdots \\ & & & & & \\ \vdots & & & \ddots & \ddots & 0 \\ 0 & \dots & & \dots & 0 & 1 \\ a_0 & a_1 & & \dots & & a_{d-1} \end{pmatrix} .$$

3. Calculer les valeurs propres de la matrice A.
4. Calculer le polynôme caractéristique de A.

Exercice 5

1. Ecrire la matrice carrée N d'ordre 6, telle que $n_{i,j} = 1$ si $j = i + 1$, 0 sinon.
2. Calculer N^k, pour $k = 1, \dots, 6$.
3. Ecrire la matrice $A = xI + N$, où x est une variable de polynôme.
4. Calculer A^k, pour $k = 1, \dots, 6$.
5. Pour $x = -2$, et $t = 0, 1, 2$, calculer $\exp(At)$.

Exercice 6 Représenter les fonctions f suivantes, en choisissant l'intervalle des abscisses et des ordonnées ainsi que le pas de discrétisation des abscisses, de façon à obtenir la représentation la plus informative possible.

1. $f(x) = 1/x$.
2. $f(x) = e^x$.
3. $f(x) = 1/\sin(x)$.
4. $f(x) = x/\sin(x)$.
5. $f(x) = \sin(x)/x$.

Exercice 7 Superposer les représentations suivantes sur le même graphique, allant de 0 à 1 en abscisse et en ordonnée.

1. La première bissectrice ($y = x$).
2. La fonction $y = f(x) = 1/6 + x/3 + x^2/2$.
3. La tangente à la fonction f au point $x = 1$.
4. Un segment vertical allant de l'axe des x au point d'intersection de la fonction f et de la première bissectrice, et un segment horizontal allant de ce point d'intersection à l'axe des y.

5. Les indications "point fixe" et "tangente", positionnées sur le graphique comme chaînes de caractères.

Exercice 8 Le but de l'exercice est de représenter sur un même graphique des familles de fonctions. On choisira le nombre de courbes, l'intervalle de représentation, les échelles en x et y ainsi que le pas de discrétisation des abscisses, de façon à obtenir la représentation la plus informative possible.

1. Fonctions $f_a(x) = x^a e^{-x}$, pour a allant de -1 à 1.
2. Fonctions $f_a(x) = 1/(x-a)^2$, pour a allant de -1 à 1.
3. Fonctions $f_a(x) = \sin(a * x)$, pour a allant de 0 à 2.

Exercice 9 Pour chacune des courbes paramétrées suivantes, on choisira un intervalle de valeurs du paramètre et un pas de discrétisation assurant une représentation complète et suffisamment lisse.

1.
$$\begin{cases} x(t) &= \sin(t) \\ y(t) &= \cos^3(t) \end{cases}$$

2.
$$\begin{cases} x(t) &= \sin(4\,t) \\ y(t) &= \cos^3(6\,t) \end{cases}$$

3.
$$\begin{cases} x(t) &= \sin(132\,t) \\ y(t) &= \cos^3(126\,t) \end{cases}$$

Exercice 10 Le but de l'exercice est de visualiser de différentes manières la surface définie par $z = f(x,y) = x\,y^2$.

1. Choisir un domaine de représentation et les pas de discrétisation, de manière à optimiser la représentation par `fsurf`.
2. Même question en utilisant `plot3d`.
3. Modifier l'échelle des couleurs pour obtenir une représentation en dégradé de gris, pour laquelle l'intensité lumineuse croît avec z.
4. Choisir un vecteur de valeurs de x. Pour chaque valeur de ce vecteur, on considère la courbe définie par $z = f(x,y)$. Représenter ces courbes sur la même fenêtre graphique.
5. Choisir un vecteur de valeurs de y. Pour chaque valeur de ce vecteur, on considère la courbe définie par $z = f(x,y)$. Représenter ces courbes sur la même fenêtre graphique.

Exercice 11 Le but de l'exercice est de visualiser un cône de différentes manières.

1. Représenter la surface d'équation $z = 1 - \sqrt{x^2+y^2}$.
2. Représenter la surface paramétrée définie par :
$$\begin{cases} x(u,v) &= u\,\cos(v) \\ y(u,v) &= u\,\sin(v) \\ z(u,v) &= 1-u\,. \end{cases}$$

3. Représenter la courbe paramétrée définie par :

$$\begin{cases} x(t) &= t\,\cos(at) \\ y(t) &= t\,\sin(at) \\ z(t) &= 1-t\ . \end{cases}$$

   (On choisira une valeur de a suffisamment grande).

4. Représenter la famille de courbes paramétrées définies par :

$$\begin{cases} x(t) &= a\,\cos(t) \\ y(t) &= a\,\sin(t) \\ z(t) &= 1-a\ . \end{cases}$$

Exercice 12 Ecrire les fonctions suivantes, sans utiliser de boucle. Toutes prennent en entrée un vecteur colonne $v = (v_i)$, un vecteur ligne $w = (w_j)$ et retournent en sortie une matrice $A = (a_{i,j})$ qui a autant de lignes que v et autant de colonnes que w. Seules les expressions des coefficients $a_{i,j}$ diffèrent.

1. `produit` : $a_{i,j} = v_i * w_j$.
2. `somme` : $a_{i,j} = v_i + w_j$.
3. `quotient` : $a_{i,j} = v_i/w_j$.
4. `echiquier` : $a_{i,j} = v_i$ si $i+j$ est pair, w_j sinon.

Exercice 13 Ecrire les fonctions suivantes, sans utiliser de boucle.

1. `insere_zeros` : Elle prend en entrée une matrice quelconque A. Elle insère une colonne de zéros après chaque colonne de `A`, et retourne en sortie la matrice modifiée (même nombre de lignes, deux fois le nombre de colonnes).
2. `alterne2_colonnes` : Elle prend en entrée deux matrices quelconques A et B, supposées de mêmes dimensions. Elle retourne la matrice formée en alternant les colonnes de A et B.
3. `alterne3_colonnes` : Même chose pour trois matrices A, B et C de mêmes dimensions.

Exercice 14 Ecrire les fonctions suivantes. Toutes prennent en entrée une fonction externe f (de $\mathbb{R}$ dans $\mathbb{R}$), et trois valeurs x_{min}, x_0 et x_{max} (supposées telles que $x_{min} \leq x_0 \leq x_{max}$).

1. `derive` : Elle calcule numériquement et représente graphiquement la dérivée de f sur l'intervalle $[x_{min}, x_{max}]$. Elle retourne la valeur approchée de $f'(x_0)$.
2. `tangente` : Elle représente la fonction f sur l'intervalle $[x_{min}, x_{max}]$, elle superpose sur le même graphique la tangente à f au point x_0, et retourne l'équation de cette tangente comme un polynôme du premier degré.
3. `araignee` : Elle représente la fonction f sur l'intervalle $[x_{min}, x_{max}]$, ainsi que la droite d'équation $y = x$ (première bissectrice). Elle calcule et retourne les 10 premiers itérés de f en x_0 ($x_1 = f(x_0), x_2 = f \circ f(x_0), \ldots$). Elle représente la suite de segments, alternativement verticaux et horizontaux, permettant de visualiser les itérations : segments joignant $(x_0, 0)$, (x_0, x_1), (x_1, x_1), (x_1, x_2), (x_2, x_2), ...

4. `newton` : Elle représente la fonction f sur l'intervalle $[x_{min}, x_{max}]$. Elle calcule et retourne les dix premiers itérés de la suite définie à partir de x_0 par la méthode de Newton : $x_1 = x_0 - f(x_0)/f'(x_0)$, $x_2 = x_1 - f(x_1)/f'(x_1)$ ... Les valeurs de la dérivée sont approchées. La fonction représente sur le même graphique les segments permettant de visualiser les itérations : segments joignant $(x_0, 0)$, $(x_0, f(x_0))$, $(x_1, 0)$, $(x_1, f(x_1))$, $(x_2, 0)$, $(x_2, f(x_2))$,...

Exercice 15 Ecrire une fonction `ech_rejet`, qui simule un échantillon par la méthode de rejet. La fonction prend en entrée les dimensions de l'échantillon à engendrer, un vecteur de modalités et un vecteur de probabilités $p = (p_i)$, $i = 1, \ldots, n$ (la distribution à simuler sur les modalités).

1. Elle calcule la constante $c = \max\{np_i\ ,\ i = 1, \ldots, n\}$, puis le vecteur $r = (n/c)p$. Pour chaque valeur à engendrer, une modalité i est tirée au hasard. Un tirage au hasard est effectué : la modalité i est conservée avec probabilité r_i, rejetée sinon. Ceci est répété jusqu'à ce que la modalité soit acceptée.
2. Tester cette fonction sur différentes distributions de probabilité, et comparer son temps d'exécution avec la fonction `ech_dist` du paragraphe 5.1.

Exercice 16 Ecrire une fonction `inversion` qui simule un échantillon d'une loi de probabilité continue par la méthode d'inversion. Les arguments de la fonction sont les deux entiers donnant la taille de la matrice à engendrer, et la fonction externe F (supposée être une fonction de répartition). Pour simuler la loi de fonction de répartition F, on appelle un nombre au hasard u sur $[0, 1]$ et on retourne le réel x tel que $F(x) = u$.

1. Ecrire la fonction. Pour résoudre l'équation $F(x) = u$, on pourra utiliser `fsolve`, ou coder une résolution par dichotomie.
2. Tester la fonction en utilisant des fonctions de répartition de lois usuelles (fonctions `cdf*`). Comparer le temps d'exécution avec celui du générateur `grand`.

Exercice 17 Ecrire une fonction `kolm_smir` pour le test de Kolmogorov-Smirnov. La fonction prend en entrée un échantillon $x = (x_i)$, $i = 1, \ldots, n$ et une fonction externe F, supposée être une fonction de répartition.

1. Elle trie l'échantillon par ordre croissant pour produire la série des statistiques d'ordre. Elle représente graphiquement les points d'abscisse (i/n) , $i = 1, \ldots, n$, et d'ordonnée les statistiques d'ordre $(x_{(i)})$, $i = 1, \ldots, n$. Elle superpose au même graphique la représentation de F.
2. Elle calcule (sans utiliser de boucle) la distance de Kolmogorov-Smirnov entre la distribution théorique et la distribution observée, par la formule :

$$D_{KS} = \max_{i=1,\ldots,n} \left\{ \left| F(x_{(i)}) - \frac{i}{n} \right| , \left| F(x_{(i)}) - \frac{i-1}{n} \right| \right\} .$$

   Elle retourne la p-valeur du test de Kolmogorov-Smirnov, comme valeur approchée de la somme :

$$p = 2 \sum_{k=1}^{+\infty} (-1)^{k+1} \exp(-2k^2 n D_{KS}^2) .$$

3. Tester la fonction `kolm_smir` sur des lois usuelles en utilisant d'abord le générateur `grand`, puis la fonction `inversion` de l'exercice précédent.

Exercice 18 Ecrire une fonction `fisher_student` prenant en entrée deux échantillons (vecteurs de nombres), et appliquant les tests de Fisher et Student à ces deux échantillons.

1. La fonction calcule les moyennes et variances empiriques des deux échantillons, ainsi que les statistiques des deux tests.
2. Elle retourne la p-valeur des deux tests (bilatéraux).

Exercice 19 Ecrire une fonction `anova` prenant en entrée trois échantillons (vecteurs de nombres), et réalisant une analyse de variance de ces échantillons.

1. La fonction calcule les moyennes et variances empiriques des trois échantillons, la variance expliquée et la variance résiduelle.
2. Elle retourne la p-valeur du test d'analyse de variance pour l'ensemble des trois échantillons, ainsi que pour chacun des trois couples d'échantillons.

Index