Utilisation de Scilab

- Démarrage de l'application → console (ligne de commande)
- Utilisation d'un éditeur (Scinotes ou autre)
- Extensions : .sce pour les scripts, .sci pour les fonctions (plus tard)
- Notion de "cahier d'expériences" pour les TPs

La console Scilab : une super-calculatrice

```
-->2*3-5/2
 ans  =

    3.5

-->cos(0)
 ans  =

    1.

-->exp(1000)
 ans  =

   Inf
```

Opérations : +, -, *, /, ^
Fonctions usuelles : log, sin, cos, tan, arctan, sinh, exp, sqrt, sign ...

... et même un peu plus

```
-->cos(%pi)
 ans  =

  - 1.

-->%i^2
 ans  =

  - 1.

-->1/%inf
 ans  =

    0.
```

Constantes : %pi, %i, %e, %inf

Les variables

```
-->a
  !--error 4
Variable non définie: a


-->a=3
 a  =

    3.

-->a^2
 ans  =

    9.
```

Attentions aux noms réservés

```
-->clear a

-->a
  !--error 4
Variable non définie: a

->exp=0;
Attention: redéfinition de la fonction: exp

-->exp(0)
       !--error 21
Index invalide.

-->clear exp; exp(1)
 ans  =

    2.7182818
```

Construction de vecteurs

```
-->v=[2,4,6,8]
 v  =

    2.    4.    6.    8.

-->v(3)
 ans  =

    6.

-->v(5)
     !--error 21
Index invalide.
```

Construction de vecteurs

```
-->v=[2,4,6,8]
 v  =

    2.    4.    6.    8.

-->v(2)=1
 v  =

    2.    1.    6.    8.

-->v(2)=[]
 v  =

    2.    6.    8.
```

Construction de vecteurs

```
-->v=[1,2]
 v  =

    1.    2.

-->w=[0,v,v,4]
 w  =

    0.    1.    2.    1.    2.    4.

-->w=[2*v,-v]
 w  =

    2.    4.  - 1.  - 2.
```

Vecteurs et matrices

```
-->v=[1;2;3]
 v  =

    1.
    2.
    3.

-->v'
 ans  =

    1.    2.    3.

-->size(v)
 ans  =

    3.    1.
```

Construction de matrices

```
-->M=[1,2;3,4]
 M  =

    1.    2.
    3.    4.

-->size(M)
 ans  =

    2.    2.

-->M(2,1)=0
 M  =

    1.    2.
    0.    4.
```

Construction de matrices

```
-->M=[1,2;3,4]
 M  =

    1.    2.
    3.    4.

-->M(1,:)
 ans  =

    1.    2.

-->M(1,:)=[]
 M  =

    3.    4.
```

Suites arithmétiques

```
-->v=0:4
 v  =

    0.    1.    2.    3.    4.

-->v=1:0.2:1.9
 v  =

    1.    1.2    1.4    1.6    1.8

-->v=1:-1:-3
 v  =

    1.    0.  - 1.  - 2.  - 3.
```

Suites arithmétiques

```
-->v=1:-1:2
 v  =

     []

-->v=linspace(1,3,5)
 v  =

    1.    1.5    2.    2.5    3.

-->v=linspace(1,-1,3)
 v  =

    1.    0.   - 1.
```

Matrices particulières

```
-->A=ones(3,2)
 A  =

    1.    1.
    1.    1.
    1.    1.

-->A=zeros(2,4)
 A  =

    0.    0.    0.    0.
    0.    0.    0.    0.
```

Matrices particulières

```
-->A=diag([3,2,1])
 A  =

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

-->A=eye(3,4)
 A  =

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

Spécification de la taille par l'exemple

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

    1.    1.    1.    1.
    1.    1.    1.    1.
    1.    1.    1.    1.

-->eye(A)
 ans  =

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

Attention : piège

```
-->A=diag([3,2,1])
 A  =

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

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

    0.    0.

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

    1.    2.
```

Deux utilisations de diag

```
->A=matrix(1:9,3,3)
 A  =

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

-->diag(A)
 ans  =

    1.
    5.
    9.

-->diag(diag(A))
 ans  =

    1.    0.    0.
    0.    5.    0.
    0.    0.    9.
```

Reformatage d'une matrice

```
->A=matrix(1:12,3,4)
 A  =

    1.    4.    7.    10.
    2.    5.    8.    11.
    3.    6.    9.    12.

-->B=matrix(A,2,6)
 B  =

    1.    3.    5.    7.    9.    11.
    2.    4.    6.    8.    10.   12.

-->B=matrix(A',3,4)
 B  =

    1.    10.   8.    6.
    4.    2.    11.   9.
    7.    5.    3.    12.
```

Boucle for

Une boucle for permet d'exécuter une suite d'instructions pour un ensemble de valeurs données d'un paramètre x. Syntaxe :

```
for x=y
  <instruction 1>
  <instruction 2>
  ...
end
```

Exemple :

```
v=0;
for x=1:10
  v=[v,x*ones(1,x)];
end
```

Eviter les boucles for

Plutôt que

```
S = 0;
for n=1:1e7
  S = S + 1/n^2;
end
```

on utilisera plutôt la commande `sum` :

```
S = sum((1:1e7).^(-2))
```

Voir aussi les commandes `prod`, `cumsum`, `cumprod`

Autre exemple : "renverser" un vecteur

```
w = [];
for i=v
  w = [i,w];
end
```

ou plus simplement

```
w = v($:-1:1);
```

Beaucoup de boucles cachées en Scilab

```
[n,p] = size(A);
for i=1:n
  for j=1:p
    B(i,j) = sin(A(i,j));
  end
end
```

est équivalent à

```
B = sin(A)
```

mais cette deuxième écriture est **beaucoup** plus efficace...

Beaucoup de boucles cachées en Scilab

```
B = A;
[n,p] = size(A);
for i=1:n
  for j=2:p
    B(i,j) = B(i,j)+B(i,j-1);
  end
end
```

est équivalent à

```
B = cumsum(A,2)
```

Supprimer les boucles for, c'est "vectoriser" un calcul

```
[n,p] = size(A);
B = zeros(n-1,p-1);
for i=1:n-1
  for j=1:p-1
    B(i,j) = A(i+1,j)+A(i,j+1)-2*A(i,j)
  end
end
```

devient

```
[n,p] = size(A);
I = 1:n-1;
J = 1:p-1;
B = A(I+1,J)+A(I,J+1)-2*A(I,J)
```

Exemple : résolution d'un système linéaire triangulaire

On veut résoudre le système linéaire $Ax = y$ où A est une matrice carrée d'ordre n inversible et **triangulaire supérieure**

$$\begin{cases} a_{11}x_1 & & & & & & & = & y_1 \\ a_{21}x_1 & + & a_{22}x_2 & & & & & = & y_2 \\ a_{31}x_1 & + & a_{32}x_2 & + & a_{33}x_3 & & & = & y_3 \\ \quad\ldots & & & & & & & & \\ a_{n1}x_1 & + & a_{n2}x_2 & + & a_{n3}x_3 & + & \ldots + & a_{nn}x_n & = & y_n \end{cases}$$

Résolution itérative pour $i = 1, 2, \ldots, n$:

$$x_i = \frac{1}{a_{n_i}} \left(y_i - \sum_{j=1}^{i-1} a_{ij} x_j \right)$$

Exemple : résolution d'un système linéaire triangulaire

$$x_i = \frac{1}{a_{n_i}} \left(y_i - \sum_{j=1}^{i-1} a_{ij} x_j \right)$$

```
for i=1:n
  S = y(i)
  for j=1:i-1
    S = S - A(i,j)*x(j)
  end
  x(i) = S/A(n,i)
end
```

Avec une seule boucle :

```
for i=1:n
  j = 1:i-1
  x(i) = (y(i)-sum( A(i,j).*x(j) ))/A(n,i);
end
```

Booléens

Scilab peut manipuler des variables numériques
mais aussi booléennes, chaînes de caractères, polynômes, etc.

Comme pour les variables numériques, les variables sont des matrices (de taille 1x1 dans le cas le plus simple)

Pour une variable booléenne, seulement deux valeurs possibles :

- **True** (T, constante %t)
- **False** (F, constante %f)

Exemple :

```
-->(1:5)>3
 ans  =

  F F F T T
```

Opérateurs de comparaison et opérateurs logiques

égalité (est-ce que $a = b$?)	`a==b` **(et non** `a=b`**)**
strictement inférieur (est-ce que $a < b$?)	`a<b`
strictement supérieur (est-ce que $a > b$?)	`a>b`
inférieur ou égal (est-ce que $a <= b$?)	`a<=b`
supérieur ou égal (est-ce que $a >= b$?)	`a>=b`
différence (est-ce que $a \neq b$?)	`a~=b`

non x (x est faux)	`~x`
x ou y (x est vrai ou y est vrai)	`x|y`
x et y (x est vrai et y est vrai)	`x&y`

tous les éléments de A sont vrais	`and(A)`
au moins un élément de A est vrai	`or(A)`

Opérateurs de comparaison et opérateurs logiques

Ces opérateurs s'appliquent terme à terme sur des matrices et des vecteurs :

```
-->A = [-3,2,1;1,-2,-2;0,1,-3]
 A  =

  - 3.    2.    1.
    1.  - 2.  - 2.
    0.    1.  - 3.

-->A>0
 ans  =

  F T T
  T F F
  F T F
```

Opérateurs de comparaison et opérateurs logiques

Comme `sum`, `cumsum`, `prod`, `cumprod`, `min`, `max`, etc.
les fonctions `and` et `or` peuvent être appliquées partiellement

Exemple :

```
-->A = rand(2,3)>0.5
 A  =

  F T F
  T T F

-->and(A,1)
 ans  =

  F T F

-->or(A,1)
 ans  =

  T T F
```

Conversion de booléens en nombres

Un booléen utilisé dans un calcul numérique sera converti en 0 (faux) et 1 (vrai)

Exemple :

```
-->v = [%t %t %f %f %t]
 v  =

  T T F F T

-->1*v
 ans  =

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

Remarque : `1*v` est équivalent à `bool2s(v)`

Exemple : valeur absolue

```
P = A>0;
B = P.*A + (1-P).*(-A);
```

est équivalent à

```
B = A;
for i = 1:length(B)
  if B(i)<0
    B(i) = -B(i)
  end
end
```

ou plus simplement à

```
B = abs(A);
```

Exemple : maximum terme à terme entre 2 matrices

```
M = (A>B).*A + (A<=B).*B;
```

est équivalent à

```
M = A;
for i = 1:length(B)
  if B(i)>A(i)
    M(i) = B(i)
  end
end
```

ou plus simplement à

```
M = max(A,B);
```

Indices booléens

Des vecteurs de booléens peuvent également être utilisés comme indices (vrai : l'indice est sélectionné, faux : il ne l'est pas)

```
-->A = [-3,2,1;-1,-2,-2;0,1,-3]
 A  =

  - 3.    2.    1.
  - 1.  - 2.  - 2.
    0.    1.  - 3.

-->A([%t %f %t],:)
 ans  =

  - 3.    2.    1.
    0.    1.  - 3.

-->A(and(A<0,2),:)
 ans  =

  - 1.  - 2.  - 2.
```

Indices booléens

Des matrices de booléens peuvent également être utilisés comme indices pour effecter des calculs conditionnels

Exemple :

```
B = A;
B(A<0) = -A(A<0)
```

est équivalent à :

```
B = A;
for i = 1:length(A)
  if A(i)<0
    B(i) = -A(i)
  end
end
```

ou plus simplement

```
B = abs(A)
```

Indices booléens

De même,

```
M = A; M(B>A) = B(B>A);
```

est équivalent à

```
M = A;
for i = 1:length(B)
  if B(i)>A(i)
    M(i) = B(i)
  end
end
```

ou plus simplement à

```
M = max(A,B);
```

Exemple : opérations sur les coefficients pairs d'une matrice

```
-->A = int(rand(3,3)*10)
 A  =

    9.    0.    4.
    4.    8.    7.
    2.    8.    1.

-->A(modulo(A,2)==0)'
 ans  =

    4.    2.    0.    8.    8.    4.

-->A(modulo(A,2)==0) = 0
 A  =

    9.    0.    0.
    0.    0.    7.
    0.    0.    1.
```

Nombre de façons d'écrire 12 comme produit de deux entiers

On construire la matrice M de terme général $M(i,j) = ij$,
puis on compte le nombre de coefficients de M qui valent 12

```
-->M = (1:12)'*(1:12);

-->sum(M==12)
 ans  =

    6.
```

Définition de fonctions Scilab

Variables en entrée : `x1,...,xm`
Variables en sortie : `y1,...,yn`

```
function [y1,...,yn] = nom_de_la_fonction(x1,...,xm)
      ...
    instructions qui définissent y1,...,yn
      ...
    (utilisation possible de variables internes)
      ...
endfunction
```

Très important :

- (re)définir les variables d'entrées d'une fonction : NON-SENS !
- les variables de sortie **doivent** être définies dans la fonction
- les variables internes ne sont pas visibles hors de la fonction
- on évite en général d'utiliser les variables externes dans une fonction

Variables de fonctions

Les variables de fonctions sont **muettes**

```
function y = f(x)
  y = x^2-sin(x)
endfunction

-->x = 1;

-->f(2)
 ans  =
    3.0907026

-->x
 x  =
    1.

-->y
  !--error 4
Variable non définie: y
```

Variables de fonctions

Une fonction peut retourner plusieurs résultats :

```
function [y,z] = f(x)
  y = x^2-sin(x)
  z = x^2
endfunction

-->[a,b] = f(2)
 b  =
    4.
 a  =
    3.0907026

-->f(2)
 ans  =
    3.0907026
```

Variables de fonctions

Un exemple très utile est celui de la fonction Scilab max (ou min) :

```
-->max([1,3,5,2,5])
 ans  =
    5.

-->[m,i] = max([1,3,5,2,5])
 i  =
    3.
 m  =
    5.

-->find([1,3,5,2,5]==5)
 ans  =
    3.    5.
```

find(x) renvoie les indices i du vecteur booléen x pour lesquels x_i est vrai.

Variables de fonctions

Les variables globales peuvent être lues **mais pas modifiées**
(ce qui serait de toute façon une mauvaise idée !)

```
function y = f(x)
  y = x^2-sin(x)
  disp(z)
  z = 1
endfunction

-->g=f(2)

    0.

 g  =
    3.0907026

-->z
 z  =
    0.
```

Fonctions vectorielles et matricielles

Lorsque l'on définit une fonction numérique par exemple, on essaie de l'écrire de façon à pouvoir faire des appels vectoriels ou matriciels

Plutôt que

```
function y = f(x)
  y = x^2-sin(x)
endfunction
```

on écrit

```
function y = f(x)
  y = x.^2-sin(x)
endfunction
```

Fonctions vectorielles et matricielles

De même, plutôt que

```
function y = f(x)
  y = x^2-sin(x)
  if y<0
    y = 1
  end
endfunction
```

on écrit

```
function y = f(x)
  y = x.^2-sin(x)
  y(y<0) = 1
endfunction
```

Récursivité

On peut définir des fonctions récursives, c'est-à-dire qui s'appellent elles-mêmes :

```
function p = fact(n)
  if n==0
    p = 1
  else
    p = n * fact(n-1)
  end
endfunction

-->fact(3)
 ans  =
    6.

-->fact(-2)
 !--error 108
```

Attention à bien vérifier que la récursivité se termine toujours !

Récursivité

On peut définir des fonctions récursives, c'est-à-dire qui s'appellent elles-mêmes :

```
function p = fact(n)
  if n<=0
    p = 1
  else
    p = n * fact(n-1)
  end
endfunction

-->fact(3)
 ans  =
    6.

-->fact(-2)
 ans  =
     1.
```

Version vectorielle

Pour des fonctions récursives la vectorisation peut être plus difficile :

```
function p = fact(n)
  p = ones(n)
  if or(n>0)
    p(n>0) = n(n>0).*fact(n(n>0)-1)
  end
endfunction

->fact([2,1;-3,4])
 ans  =

    2.    1.
    1.    24.
```

Elimination de la récursivité

Dans certains cas il est possible de transformer une fonction récursive en fonction non récursive :

```
function p = fact(n)
  p = prod(1:n)
endfunction
```

Pour la forme vectorielle, on ne peut pas éviter le recours à une boucle :

```
function p = fact(n)
  p = ones(n)
  while or(n>0)
    p = p.*max(1,n)
    n = n-1
  end
endfunction
```

Exemple de la suite de Syracuse

Suite $(u_n)_{n\geq 0}$ définie par $u_0 \in \mathbb{N}$ et la récurrence

$$\forall n \geq 1, \quad u_n = \begin{cases} u_{n-1}/2 & \text{si } u_n \text{ est pair,} \\ 3u_{n-1}+1 & \text{sinon.} \end{cases}$$

```
function un = syracuse(u0,n)
  if n<=0
    un = u0
  else
    x = syracuse(u0,n-1)
    if modulo(x,2)==0
      un = x/2
    else
      un = 3*x+1
    end
  end
endfunction
```

Exemple de la suite de Syracuse

Version non récursive :

```
function un = syracuse(u0,n)
  un = u0
  for k=1:n
    if modulo(un,2)==0
      un = un/2
    else
      un = 3*un+1
    end
  end
endfunction
```

Exemple de la suite de Syracuse

Calcul du temps de vol
(étant donné u_0, l'indice n minimal tel que $u_n = 1$) :

```
function n = temps_de_vol(u0)
  n = 0
  while u0>1
    n = n+1
    if modulo(u0,2)==0
      u0 = u0/2
    else
      u0 = 3*u0+1
    end
  end
endfunction
```

On ne sait pas démontrer que cette fonction termine pour tout u_0 !

Rappels sur les structures de contrôle

```
if <condition>
  ...
end

if <condition>
  ...
else
  ...
end

if <condition>
  ...
elsif <condition>
    ...
  else
    ...
  end
end
```

Rappels sur les structures de contrôle

```
for <variable> = ...
  ...
end


while <condition>
  ...
end


select <variable>
case <valeur>
  ...
case <valeur>
  ...
else
  ...
end
```

Exemple avec l'instruction select

```
function dalton(i)
  select i
  case 1
    disp("Joe")
  case 2
    disp("William")
  case 3
    disp("Jack")
  case 4
    disp("Averell")
  else
    disp("Indice incorrect !")
  end
endfunction

-->dalton(4)
 Averell

-->dalton(1.5)
 Indice incorrect !
```

Multiplication matricielle : associative mais...

Soient `A`, `B`, `C` des matrices de tailles $m \times n$, $n \times p$, $p \times q$

Pour calculer le produit matriciel `A*B*C`, il est mathématiquement équivalent d'écrire `(A*B)*C` ou `A*(B*C)`. Est-ce indifférent lors d'un calcul effectif en Scilab ?

$\rightarrow$ Non car le calcul "classique" de `(A*B)*C` requiert $mnp + mpq = mp(n+q)$ multiplications élémentaires, alors que le calcul de `A*(B*C)` requiert $npq + mnq = nq(m+p)$ multiplications.

Exemple :
`A` matrice 10×100, `B` matrice 100×10, `C` matrice 10×100

$mp(n+q) = 100 \times 200 = 20.000$
$nq(m+p) = 10000 \times 20 = 200.000$

Erreurs d'arrondi

```
-->%e
 %e   =  2.7182818

-->sqrt(-1)
ans  =  i

-->(2+%i*3)*(1-%i)
 ans  =  5. + i

-->sin(%pi/2)
 ans  =  1.
```

```
-->format("v",25)

-->1/3
   ans =   0.333333333333333148296

-->100000/3
 ans  =   33333.333333333335758653

-->tan(atan(1))
 ans  =   0.999999999999998889777
```

Sur l'ordinateur, on a une précision finie et un temps limité !

Erreurs d'arrondi

La mémoire disponible étant finie, on ne peut pas représenter les nombres réels qui ont un développement décimal infini.

On peut représenter le rationnel 1/3,
mais pas son écriture décimale.

On ne peut représenter π que par un nombre fini de décimales.

Un représentation fréquente est celle en *virgule flottante* des nombres réels :
$f = (-1)^s \cdot 0, d_{-1}d_{-2}\ldots d_{-r} \cdot b^j$,
$m \leq j \leq M$ et $0 \leq d_{-i} \leq b-1$;
où s est le *signe*, $d_{-1}d_{-2}\cdots d_{-r}$ la *mantisse*, b la *base* et r est le nombre de *chiffres significatifs*.

La *précision machine* pour un flottant est la distance entre 1 et le nombre flottant immédiatement plus grand $1 + b^{1-r}$.
En **Scilab** `%eps = 2.220D-16`.

Erreurs d'arrondi

Le résultat d'un calcul n'est pas nécessairement représentable dans la machine : on est obligé d'arrondir.
Entre `1` et `1+%eps` aucun nombre de la droite réelle n'est représentable (et de même entre `x` et `x*(1+%eps)`).

```
-->format("v",25)
-->%eps
 %eps   =     0.00000000000000002220446
-->a=1+%eps
 a      =     1.00000000000000002220446
-->b=2+%eps
 b      =     2.
-->b=2*(1+%eps)
 b      =     2.00000000000000004440892
-->c=(b+2)-2
 c      =     2.
```

En effectuant un *grand* nombre d'opérations, les *erreurs* d'arrondi vont finir par *perturber* le résultat final.

Exceptions

Scilab possède trois modes pour la gestion d'*exceptions*, grâce à `ieee()` on définit ces modes :

- mode `ieee(0)` : arrêt d'exécution avec message d'erreur ;
- mode `ieee(1)` : émission de "warnings" et résultats avec `Inf` et `NaN` ;
- mode `ieee(2)` : aucun message, résultats avec `Inf` et `NaN`.

Ceci permet de gérer des problèmes numériques sans arrêter l'exécution d'un programme.

Gestion d'erreurs

```
                                  |
-->ieee(0);1/0                    | -->x=linspace(-1,1,201);
            !--error 27           | -->x(101)
division by zero...               |  ans  = 0.
                                  |
-->ieee(1);1/0                    | -->ieee(0)
Warning :division by zero...      | -->y=(1)./x;
                                  |       !--error 27
 ans  =    Inf                    |       division by zero...
                                  |
-->ieee(2);1/0, log(0), 0/0       | -->ieee(2)
 ans  =    Inf                    | -->y=(1)./x; y(101)
 ans  =   -Inf                    |  ans  = Inf
 ans  =    Nan                    | -->plot2d(x,y)//affichage
                                  |               //sans y(101)
```