Informatique et algorithmique avec le logiciel Scilab en ECS 1ère année

BÉGYN Arnaud

22/01/2014

Table des matières

1 Généralités sur l'algorithmique et le logiciel Scilab

1.1 Introduction

Un *algorithme* est une suite finie de règles à appliquer, dans un ordre déterminé, à un nombre fini de données, pour arriver en un nombre fini d'étapes à un certain résultat, et cela indépendamment des données . Un algorithme doit donc fournir de manière **certaine** le résultat escompté, en temps **fini**.

L'algorithme mathématique le plus connu est celui d'Euclide : il permet de trouver le plus grand diviseur commun à deux entiers.

La structure d'un algorithme est celle d'une recette de cuisine : on commence par énumérer tous les ingrédients nécessaires, puis on décrit les étapes successives à faire. Si l'algorithme est bon, un merveilleux gâteau sort du four !

En fait tout algorithme se décompose en trois parties. La première partie concerne les **entrées** : on liste toutes les données nécessaires. La deuxième partie regroupe toutes les instructions. Le troisième partie est la **sortie** du résultat.

L'intérêt majeur de disposer d'algorithmes de résolution de problèmes est de permettre l'automatisation : on peut déléguer à une machine l'exécution de tâches routinières conduisant à la solution.

La traduction de l'algorithme en vue de son exécution par une machine pose deux problèmes distincts :

a) le découpage en tâches suffisamment simples et sans ambiguïté pour qu'une machine sache les exécuter correctement :

b) la traduction de ces tâches dans un **langage** compréhensible par la machine.

Une méthode d'élaboration d'un bon algorithme est appelé méthode descendante (ou top-down). Elle consiste à considérer un problème dans son ensemble, à préciser les données fournies et les résultats à obtenir, puis à décomposer le problème en plusieurs sous-problèmes plus simples, qui seront traités séparément et éventuellement décomposés eux-mêmes de manière plus fine.

Prenons un exemple : imaginons un robot ménager à qui nous devons fournir un algorithme lui permettant de préparer une tasse de café soluble. Une première version de l'algorithme pourrait être :

(1) faire bouillir de l'eau
(2) mettre le café dans la tasse
(3) ajouter l'eau dans la tasse

Les étapes de cet algorithme ne sont probablement pas assez détaillées pour que le robot puisse les interpréter. Chaque étape doit donc être affinée en une suite d'étapes plus élémentaires, chacune étant spécifiée d'une manière plus détaillée que dans la première version. Ainsi l'étape (1) peut être affinée en :

(1.1) remplir la bouilloire d'eau
(1.2) brancher la bouilloire sur le secteur
(1.3) attendre l'ébullition
(1.4) débrancher la bouilloire

De même (2) pourrait être affinée en :

(2.1) ouvrir le pot à café
(2.2) prendre une cuillère à café
(2.3) plonger la cuillère dans le pot
(2.4) verser le contenu de la cuillère dans la tasse

(2.5) fermer le pot à café

Et (3) pourrait être affinée en :
(3.1) verser de l'eau dans la tasse
(3.2) s'arrêter lorsque celle-ci est pleine

Certaines étapes étant encore trop complexes et sans doute incompréhensibles par notre robot, il faut les affiner davantage. Ainsi l'étape (1.1) peut nécessiter les affinements suivants :
(1.1.1) mettre la bouilloire sous le robinet
(1.1.2) ouvrir le robinet
(1.1.3) attendre que la bouilloire soit pleine
(1.1.4) fermer le robinet

Quand il procède à des affinements des différentes étapes, le concepteur d'un algorithme doit naturellement savoir où s'arrêter. Autrement dit, il doit savoir quand une étape est compréhensible par le robot ou la machine. Ceci dépend du langage de programmation utilisé.

Un programme est alors la traduction d'un algorithme en langage compréhensible par la machine. Il existe de nombreux langages compréhensibles par un ordinateur. Chacun possède ses particularités propres, mais les principes de base sont toujours plus ou moins les mêmes.

Nous n'étudierons que le langage de Scilab. C'est un logiciel libre (c'est-à-dire gratuit), dédié au calcul et aux simulations numériques. Nous pourrons ainsi illustrer et vérifier numériquement des résultats démontrées formellement dans le cours de mathématiques.

1.2 L'environnement du logiciel Scilab

Scilab se lance (sous windows) par un double clique sur l'icône ad hoc, située généralement sur le bureau. On obtient alors l'ouverture d'une fenêtre mais il faut attendre l'apparition d'une flèche --> avant d'utiliser le logiciel.

La fenêtre principale est composée de quatre sous-fenêtres :

• Console Scilab : c'est la fenêtre dans laquelle on tape les commandes pour exécution immédiate, et dans laquelle sont affichées les réponses.

• Navigateur de variables : contient le nom, la taille et le type de variables reconnues par la console.

• Navigateur de fichiers : précise le répertoire courant (celui dans lequel on travaille), et permet de naviguer dans le système de fichiers.

• Historique des commandes : contient toutes les lignes de commande validées dans la console.

Le menu Édition permet de nettoyer ces sous-fenêtres, lorsqu'elles deviennent surchargées.

La console permet d'utiliser Scilab comme une calculatrice évoluée. Chaque ligne de commande commence par l'invite --> (qui est mise automatiquement par le logiciel). Cette invite n'apparaît pas lors de l'exécution d'instructions ou lorsque le logiciel attend une saisie de l'utilisateur au clavier.

Une instruction tapée dans la console est **immédiatement exécutée** après avoir validé à l'aide la touche entrée.

On a ensuite deux possibilités :
- si la ligne de commande se termine par le symbole `;`, l'instruction est exécutée mais le résultat n'est pas affiché à l'écran (ce qui peut être utile si le résultat est un tableau de 10000 valeurs numériques). On dit que l'exécution s'est faite **sans écho**.
- si la ligne de commande ne se termine pas par le symbole `;` (elle se termine alors par un espace ou par le symbole `,`) l'instruction est exécutée et le résultat est affiché.

Une ligne de commande peut être modifiée tant qu'elle n'est pas validée. Une fois validée, on ne peut par contre plus la corriger. Cependant, on peut rappeler toute ligne de commande précédemment validée dans la console en utilisant les flèches haut et bas du clavier : $\boxed{\uparrow}$ et $\boxed{\downarrow}$.

Si une instruction produit un résultat et ne comporte pas d'instruction d'affectation (c'est-à-dire que le résultat n'est pas enregistré dans une variable), alors Scilab enregistre le résultat par défaut dans la variable `ans` (abréviation de answer).

Le logiciel Scilab dispose d'un *Navigateur d'aide* accessible via l'onglet $\boxed{?}$ ou la touche $\boxed{F1}$. Si l'on connaît le nom de la fonction pour laquelle on recherche de l'aide, on peut aussi utiliser dans la console l'instruction `help` suivi du nom de la fonction. L'instruction `help` toute seule donne une autre façon d'accéder au navigateur d'aide.

Les explications de l'aide sont souvent très techniques : en première approche, il ne faut donc pas hésiter à regarder les exemples donnés en bas de chaque page d'aide.

2 Scilab en tant que calculatrice évoluée

2.1 Manipulation des nombres avec Scilab

Tous les nombres manipulés sont des nombres complexes de la forme $x + iy$ où x et y sont deux nombres décimaux. Par défaut les nombres décimaux sont donnés sous forme décimale avec 7 chiffres significatifs. Le nombre complexe i tel que $i^2 = -1$ est représentée dans Scilab par la syntaxe `%i`.

Les nombres décimaux utilisés dans Scilab sont compris entre 2.10^{-38} et 2.10^{38}, avec une précision de $2,22.10^{-16}$.

Les opérations sur les nombres se font à l'aide des symboles suivants :
- `+` (addition)
- `-` (soustraction)
- `*` (multiplication)
- `/` (division)
- `^` (puissance)

Dans une ligne de calcul, l'ordre utilisé par Scilab est le même que celui que nous utilisons en mathématiques, c'est-à-dire par ordre décroissant de priorité : `^`, ensuite `*` ou `/`, puis `+` ou `-`.
Pour modifier cette ordre, on utilise des parenthèses.

Exemple Par exemple :

```
-->2*3-5
 ans  =
    1.
-->2-3*5
 ans  =
  - 13.
-->(2-3)*5
 ans  =
  - 5.
```

Le nombre π s'utilise via le symbole `%pi` et le nombre e via `%e`. Ce ne sont bien sûr que des approximations de ces nombres.

On dispose aussi des fonctions suivantes déjà définies dans le logiciel :

`log`	(logarithme népérien ln)
`exp`	(exponentielle)
`floor`	(partie entière)
`abs`	(valeur absolue)
`sqrt`	(racine carrée)
`sin`	(sinus)
`cos`	(cosinus)

2.2 Variables et affectations

Les variables sont des « cases mémoires » dans laquelle une valeur est enregistrée. Cette valeur peut être modifiée tout au long du programme. Une variable est identifiée par un nom (ou identificateur). Ce nom ne doit comporter ni espace, ni accent, ni apostrophe et doit obligatoirement commencer par une lettre. Scilab distingue la casse, c'est-à-dire les majuscules et les minuscules.

Lorsqu'on modifie la valeur d'une variable, on dit qu'on lui **affecte** une valeur. L'**affectation** à la variable `Nom` de la valeur `Valeur` se fait avec la syntaxe suivante :

`-->Nom=Valeur` ou `-->Nom=Valeur;` (sans écho)

S'il n'existe pas de variable appelée `Nom`, cette instruction **crée une variable** `Nom` et lui affecte `Valeur`. S'il existe déjà une variable `Nom`, cette instruction **efface l'ancienne affectation et la remplace** par `Valeur`.

Lorsqu'une variable est créée, son nom apparaît dans le *Navigateur de variables* avec une icône qui indique la nature de son contenu.

⚠ Il faut bien respecter la syntaxe : ce qui se trouve **à gauche** du symbole = est modifié/créé par ce qui se trouve **à droite** du =.

⚠ Si on utilise un nom de variable non défini préalablement dans Scilab à droite de = on provoque une erreur.

Exemple Affectation avec écho :

```
-->x=2*3
 x  =
    6.
```

et sans écho :

```
-->x=2*3;
```

⚠ La valeur d'une variable est modifiée tout au long du programme.
Par exemple quelle est la valeur de `x` à la fin des instructions suivantes ?

```
-->x=2*3;
-->y=4;
-->y=2*y+1;
-->x=x*y;
```

Pour connaître la valeur affectée à une variable existante, il suffit de taper son nom dans la console et de valider. Par exemple sur l'exemple précédent :

```
-->x
 x  =
    54.
```

Pour connaître toutes les variables créées, on utilise la commande `who` (ou `whos` dont l'affichage est différent).

La commande `clear` permet d'effacer toutes les variables. La commande `clear x` permet d'effacer uniquement la variable `x`.

2.3 Les chaînes de caractères

C'est un **type de données** (comme l'est le type nombre), qui permet de manipuler du texte avec Scilab.

On appelle **chaîne de caractères** (« string » en anglais), une suite de *caractères alphanumériques* (les caractères disponibles sur le clavier d'un ordinateur).

Dans le langage Scilab, les chaînes de caractères sont délimitées par des apostrophes `'`. Lorsqu'une chaîne de caractère contient déjà le caractère `'`, il faut le doubler.

Exemple Par exemple :

```
--> 'Lorsqu''une'
 ans =
    Lorsqu'une
```

On peut affecter une chaîne de caractères à une variable.

Exemple Par exemple :

```
--> a = 'Scilab'
  a =
    Scilab
```

⚠ Scilab distingue une chaîne de chiffres du nombre qu'elle représente. Par exemple `'12'` est différent de `12` : le premier est du type chaîne de caractères et le second du type nombre.

Exemple Par exemple :

```
--> Txt = '12+13'
 Txt =
    12+13
```

L'instruction `Txt='12+13'` affecte à la variable `Txt` la valeur `12+13` (et non pas 25) qui est une chaîne de cinq caractères. L'icône associée à `Txt` dans le navigateur de variables indique qu'on est en présence d'une chaîne de caractères.

Le nombre de caractères d'une chaîne `A` s'obtient avec l'instruction `length(A)`.

Exemple Par exemple :

```
--> length('toto')
 ans =
    4.
```

Si `A` est une chaîne de caractères, alors `part(A,i)` retourne son i-ième caractère (avec la convention qu'ils sont numérotés à partir de 1).

Exemple Par exemple :

```
--> A = '13=XIII'
  A =
    13=XIII

--> part(A,1)
 ans =
    1

--> part(A,2)
 ans =
    3

--> part(A,3)
 ans =
    =

--> part(A,8)
 ans =
```

⚠ Remarquer que le caractère 3 se distingue dans la réponse de Scilab du chiffre 3 : en effet, ce dernier se note avec un point 3. et le premier sans point.

⚠ Remarquer que la dernière instruction n'a pas produit d'erreur mais a donné le caractère vide, noté ''. Il ne faut pas le confondre avec le caractère espace ' '.

Plus généralement, l'instruction part(A,i:j) donne la sous-chaîne extraite de A, formée des caractères entre les positions i et j (incluses).

Exemple Par exemple :

```
--> Txt = '12+13';

--> part(Txt,2:4)
 ans =
    2+1
```

L'opérateur addition + permet de coller deux chaînes de caractères : c'est l'opération de **concaténation**.

Exemple Par exemple :

```
--> A = 'Toto'
  A =
    Toto

--> B = 'Bonjour '
  B =
    Bonjour

--> B + A
 ans =
    Bonjour Toto
```

2.4 Les variables booléennes

Le résultat d'une expression logique (c'est-à-dire « vrai » ou « faux ») peut être stocké dans une variable, appelée alors **variable booléenne**.

Les booléens sont %t (true) et %f (false) pour les saisies et T et F pour l'affichage.

Les variables booléennes sont le résultat d'opérations logiques comme la comparaison, la négation etc ...

Exemple Par exemple :

```
--> a = %f
  a =
    F
```

Les symboles de comparaison sont :
- `==` (test d'égalité) ⚠ à ne pas confondre avec l'affectation `=` !
- `~=` (différent)
- `<` (inférieur strict)
- `<=` (inférieur ou égal)
- `>` (supérieur strict)
- `>=` (supérieur ou égal)

Exemple Par exemple :

```
--> 3 == 5
 ans =
    F

--> 3 = 5
Attention : Utilisation obsolète de '=' à la place de '=='.
   !


 ans =

    F
```

On voit que Scilab a remarqué une erreur dans la commande `3=5`, l'a signalé par un message, et a ensuite corrigé l'erreur et donné le bon résultat. Il est préférable d'éviter cette confusion de commandes, même si Scilab peut parfois la corriger.

On peut également former des expressions logiques plus compliquées :
- `~` (négation logique)
- `&` (« et » logique)
- `|` (« ou » inclusif) ⚠ c'est celui qu'on utilise en mathématiques.

Sous Mac, le symbole `|` s'obtient par la combinaison de touches Alt+Shift+L.

Exemple Par exemple :

```
--> ( 3 == 5 ) & ( 3 == ( 2 + 1 ) )
 ans =
    F

--> ( 3 == 5 ) | ( 3 == ( 2 + 1 ) )
 ans =
    T
```

3 Les matrices en Scilab

Une **matrice** $n \times p$ est un tableau à n lignes et p colonnes, dont les coefficients (les valeurs de chaque case) sont des nombres. Scilab est un langage fondé sur la manipulation des matrices, il faut donc les utiliser au maximum. Par exemple un nombre décimal est considéré par Scilab comme une matrice 1×1.

Lorsque $n = 1$ (une seule ligne), la matrice est appelée **vecteur ligne**, et lorsque $p = 1$ (une seule colonne), elle est appelée **vecteur colonne**.

Nous allons voir plusieurs façons de définir une matrice : en effet, une matrice 2×3 peut être définie en donnant la valeur de ses 6 coefficients, mais cela risque de devenir très fastidieux avec les 10^{10} coefficients d'une matrice 10000×10000 !

3.1 Vecteurs lignes

• *Définition par extension = définition par la donnée de la liste de ses éléments*

On donne la liste des éléments, **entre crochets**, séparés par des **virgules**. C'est la manière la plus simple mais cela n'est possible qu'avec des matrices de petite dimension (n et p petits).

Sous Mac, les crochets s'obtiennent par les combinaisons de touches Alt+Shift+(et Alt+Shift+)
.

Exemple Par exemple :

```
--> X = [ 2 , %pi , 0 , -37 ]
  X  =
     2.    3.1415927    0.  - 37.
```

• *Définition par une progression arithmétique*

Il suffit de donner une valeur initiale, une valeur à atteindre, et la raison de la progression arithmétique.

L'instruction `--> X = m : h : M` crée un vecteur ligne X dont le premier coefficient est m, le second $m+h$, le troisième $m+3h$, …, le $(k+1)$-ième est $m+kh$, …, et le dernier est $m+kh$ avec k tel que $m+kh \leq M$ et $m+(k+1)h > M$, c'est-à-dire que le dernier coefficient est le plus grand possible de la forme $m+kh$ (avec $k \in \mathbb{N}$), mais sans dépasser M.

⚠ Si ($M < m$ et $h > 0$) ou ($M > m$ et $h < 0$) (cas « absurdes »), le logiciel **ne renvoie pas d'erreur** mais crée une matrice vide, c'est-à-dire une matrice 0×0. Elle est notée `[]`.

⚠ Retenir que `--> X = m : h : M` ne crée que des **vecteurs lignes**. Ce sera important dans la suite.

Exemple Par exemple :

```
--> X = 1 : 0.3 : 2
 X   =
     1.    1.3    1.6    1.9

--> X = 8 : -1 : 6
 X   =
     8.    7.    6.

--> X = 6 : -1 : 8
 X   =
      []
```

Lorsque $h = 1$, on peut omettre de le préciser.

Exemple Par exemple :

```
--> X = -1 : 4.5
 X   =
   - 1.    0.    2.    3.    4.
```

• *Définition à l'aide d'une fonction d'initialisation*

L'instruction `--> X = linspace(a,b,n)` crée un vecteur ligne de n coefficients régulièrement espacés entre a et b, c'est-à-dire une subdivision de l'intervalle $[a, b]$ en n points, et donc $n-1$ sous-intervalles.

Elle donne le même résultat que `--> X = a : (b-a)/(n-1) : b`

⚠ Retenir que `linspace` ne crée que des **vecteurs lignes**.

L'instruction `--> X = zeros(1,n)` crée un vecteur ligne de n coefficients tous nuls.

Exemple `--> X= zeros(1,3)` donne le même résultat que `--> X = [ 0 , 0 , 0 ]` .

L'instruction `--> X = ones(1,n)` crée un vecteur ligne de n coefficients tous égaux à 1.

Exemple `--> X= ones(1,4)` donne le même résultat que `--> X = [ 1 , 1 , 1 , 1 ]` .

3.2 Vecteurs colonnes

• *Définition par extension*

On donne la liste des éléments, **entre crochets**, séparés par des **points virgules**.

Exemple Par exemple :

```
--> X = [ 2 ; %pi ; 0 ; -37 ]
  X  =
     2.
     3.1415927
     0.
   - 37.
```

• *Définition à l'aide d'une fonction d'initialisation*

L'instruction `--> X = zeros(n,1)` crée un vecteur colonne de n coefficients tous nuls.

L'instruction `--> X = ones(n,1)` crée un vecteur ligne de n coefficients tous égaux à 1.

3.3 Matrices : cas général

• *Définition par extension*

On donne la liste des éléments, **entre crochets**, séparés par des **virgules**, chaque ligne étant séparé par des **points virgules**.

Exemple Par exemple :

```
--> D = [ 1 , 2 ,4 ; 2 , 3 , 5 ]
  D  =
     1.    2.    4.
     2.    3.    5.

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

• *Définition à l'aide d'une fonction d'initialisation*

L'instruction `--> M = zeros(n,p)` crée une matrice de dimension $n \times p$ dont tous les coefficients sont nuls.

L'instruction `--> M = ones(n,p)` crée une matrice de dimension $n \times p$ dont tous les coefficients sont égaux à 1.

L'instruction `--> M = eye(n,p)` crée une matrice de dimension $n \times p$ dont tous les coefficients sont égaux à 0, sauf ceux de la diagonale qui sont égaux à 1.

3.4 Manipulation des coefficients d'une matrice

Si `X` est un vecteur, `X(i)` retourne le i-ième coefficient (ils sont numérotés à partir de 1).

On peut modifier ce coefficient grâce à l'instruction `X(i) =` et à droite de `=` on peut mettre soit une valeur numérique, soit une variable déjà créée auparavant.

⚠ Si `X` a n coefficient et $i \geq n+1$ ou $i \leq 0$ alors l'instruction `Variable = X(i)` renvoie un message d'erreur.

⚠ Si `X` a n coefficient et $i \geq n+1$ alors l'instruction `X(i) =` une variable ou une valeur numérique ne renvoie pas de message d'erreur, mais agrandi le vecteur `X` avec des coefficients nuls jusqu'au i-ième qui prend lui la valeur demandée. Par contre, si $i \leq 0$ on obtient un message d'erreur.

Le nombre de coefficient d'un vecteur `X` s'obtient avec l'instruction `length(X)`.

Exemple Par exemple :

```
--> X = [ 1 , 2 ]
  X  =
     1.    2.

--> X(2)
  ans  =
     2.

--> X(5)
      !--error 21
Index invalide

--> X(5) = 2.3
  X =
    1.    2.    0.    0.    2.3

--> X(-1) = 3.
      !--error 21
Index invalide

--> a = X(6)
      !--error 21
Index invalide

--> a = X(5)
  a =
    2.3

--> X(3) = 3 ; X
  X =
    1.    2.    3.    0.    2.3

--> length(X)
 ans =
    5.
```

L'apostrophe permet de transformer un vecteur ligne en vecteur colonne, et inversement : c'est l'opération de **transposition**. Combinée avec les instructions `X = m : h : M` et `linspace`, on obtient une nouvelle méthode pour créer des vecteurs colonnes.

Exemple Par exemple

```
--> X = 1 : 3
  X =
    1.    2.    3.
```

```
--> Y = X'
  Y =
    1.
    2.
    3.
```

Si `M` est une matrice de dimension $n \times p$, l'instruction `size(M)` renvoie le **vecteur ligne** `[n,p]`. Pour définir deux variables n et p égales respectivement aux nombres de lignes et de colonnes de `M`, on utilise l'instruction `[n,p]=size(M)`.

Si `M` est une matrice de dimension $n \times p$, l'instruction `length(M)` renvoie son nombre de coefficients np.

Le coefficient de `M` situé ligne i colonne j s'écrit `M(i,j)`. Il faut que i et j soient dans $\mathbb{N}^*$ sinon on obtient une erreur.

Si `M` est de dimension $n \times p$, et si $i > n$ ou $j > p$ alors invoquer `M(i,j)` à droite de = donne une erreur. Par contre invoquer `M(i,j)` à gauche de = **agrandit le tableau**, les coefficients manquants étant initialisés à 0.

Exemple Par exemple :

```
--> X = [ 1 ; 2 ]
  X =
    1.
    2.

--> X(3) = -1
  X =
    1.
    2.
   -1.

--> X(2,2)=36
  X =
    1.    0.
    2.   36.
   -1.    0.

--> size(X)
 ans =
    3.    2.

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

--> T = C(1,1) + C(2,2) + C(3,3)
  T =
   15.
```

```
--> C(2,1) = 3
  C =
    1.    2.    3.
    3.    5.    6.
    7.    8.    9.
```

Si `M` est une matrice, alors sa i-ième ligne s'obtient avec l'instruction `M(i,:)`, et la j-ième colonne par `M(:,j)`.

Si `M` est une matrice de dimension $n \times p$ alors la sous-matrice obtenue entre les lignes ℓ_1 et ℓ_2, et entre les colonnes c_1 et c_2, s'obtient avec l'instruction `M(l1:l2,c1:c2)`.

Exemple Par exemple :

```
--> C = [ 1 , 2 , 3 ; 4 , 5 , 6 ; 7 , 8 , 9 ; 10 , 11 ,12 ]
  C =
    1.    2.    3.
    4.    5.    6.
    7.    8.    9.
   10.   11.   12.

--> B = C(1:3,2:3)
  B =
    2.    3.
    5.    6.
    8.    9.

--> D= B(2,:)
  D =
    5.    6.
```

On peut juxtaposer deux matrices : c'est l'opération de **concaténation** :

- si A et B sont deux matrices ayant le **même nombre de lignes** alors l'instruction `C = [ A , B ]` crée une matrice `C` obtenue en juxtaposant **horizontalement** (c'est-à-dire ligne par ligne) les matrices A et B.

- si A et B sont deux matrices ayant le **même nombre de colonnes** alors l'instruction `D = [ A ; B ]` crée une matrice `D` obtenue en juxtaposant **verticalement** (c'est-à-dire colonne par colonne) les matrices A et B.

Exemple Par exemple :

```
--> A = zeros(3);

--> B = ones(3,2);

--> C = [A,B]
  C =
    0.    0.    0.    1.    1.
    0.    0.    0.    1.    1.
    0.    0.    0.    1.    1.
```

Exemple Par exemple :

```
--> A = zeros(3);

--> B = eye(2,3);

--> D = [A;B]
  D =
    0.    0.    0.
    0.    0.    0.
    0.    0.    0.
    1.    0.    0.
    0.    1.    0.
```

Pour obtenir le **rang** d'une matrice, il suffit d'utiliser l'instruction `rank`.

```
--> M = [ 1 , 1 ; 1 , 1 ];

--> rank(M)
 ans =
    1.
```

La **transposée** d'une matrice s'obtient avec l'apostrophe.

```
--> A = [1 , 2 , 3 ; 4 , 5 , 6 ]
  A =
    1.    2.    3.
    4.    5.    6.

->> A'
 ans =
    1.    4.
    2.    5.
    3.    6.
```

3.5 Opérations sur les matrices

Si `A` et `B` sont deux matrices ayant la même dimension et `x` est un nombre complexe, on peut faire les opérations suivantes coefficient par coefficient :

• **Addition coefficient par coefficient :** l'instruction `--> C = A .+ B` définit une matrice `C` de même dimension que `A` et `B` et telle que `C(i,j) = A(i,j) + B(i,j)`. On peut aussi utiliser « + » à la place de « .+ ».

• **Multiplication coefficient par coefficient :** l'instruction `--> D = A .* B` définit une matrice `D` de même dimension que `A` et `B` et telle que `D(i,j) = A(i,j) * B(i,j)`.

⚠ L'instruction « * » ne correspond pas à la même opération (voir produit matriciel).

• **Multiplication des coefficients par un nombre :** l'instruction `--> E = x.*A` définit une matrice `E` de même dimension que `A` et telle que `E(i,j) = x * A(i,j)`. On peut aussi utiliser « * » à la place de « .* ».

• **Addition des coefficients par un nombre :** l'instruction `--> F = x.+A` définit une matrice `F` de même dimension que `A` et telle que `F(i,j) = x + A(i,j)`. On peut aussi utiliser « + » à la place de « .+ ».

• **Division coefficient par coefficient :** si aucun des coefficients de B n'est égal à zéro, l'instruction `--> G = A ./ B` définit une matrice `G` de même dimension que `A` et `B` et telle que `G(i,j) = A(i,j) / B(i,j)`.

⚠ L'instruction « / » ne correspond pas à la même opération (elle n'est pas au programme).

Pour tout réel k, si toutes les puissances `A(i,j)^k` sont définies, alors :

• **Puissance k-ième coefficient par coefficient :** l'instruction `--> H = A .^ k` définit une matrice `H` de même dimension que `A` et telle que `H(i,j) = A(i,j)^k`.

⚠ L'instruction « ^ » ne correspond pas à la même opération.

En général, si `f` est une fonction prédéfinie dans Scilab et opérant sur les nombres (`cos`, `sqrt`, ...), alors :

• **Fonction coefficient par coefficient :** l'instruction `--> I = f(A)` définit une matrice `I` de même dimension que `A` et telle que `I(i,j) = f(A(i,j))`.

Exemple Par exemple :

```
--> A = [ 4 , 9 , 25 ; -1 , 4 ,0 ] ;

--> B = A .^ (0.5)
  B =
    2.    3.    5.
    i.    2.    0.

--> C = floor(A/2)
  C =
    2.    4.   12.
   -1.    2.    0.
```

Scilab permet aussi d'utiliser le calcul matriciel vu en cours de mathématiques. Si `A` et `B` sont deux matrices telles que le nombre de colonnes de A est égal au nombre de lignes de `B`, on peut les multiplier.

• **Produit matriciel et puissance d'une matrice :** l'instruction `--> J = A * B` définit une matrice `J` égale au produit matriciel de `A` et `B`, et l'instruction `--> L = A^k` définit une matrice `L` égale à la puissance k-ième de `A`.

Si `A` est une matrice de dimension $n \times n$, l'instruction `inv(A)` donne l'inverse de la matrice `A` si elle existe et une erreur sinon (on peut aussi utiliser `A^(-1)`.

Exemple Par exemple :

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

--> B = ones(3);

--> A * B
 ans =
      6.    6.    6.
     15.   15.   15.
     24.   24.   24.

--> C=[ 1 , -1 , -1 ; -1 , 1 , -1 ; -1 , -1 , 1 ]; inv(C)
 ans =
      0.   - 0.5   - 0.5
    - 0.5    0.    - 0.5
    - 0.5  - 0.5     0.
```

3.6 Matrice vide

Il existe dans Scilab une matrice de dimension 0×0, appelée **matrice vide**, et notée `[]`.

⚠ Dans Scilab, la matrice vide `[]` est égale à la chaîne de caractères vide `''`.

Elle interagit avec n'importe quelle matrice `A` de la manière suivante :

```
[] + A = A + [] = A
[] * A = A * [] = []
```

On peut l'utiliser pour détruire des lignes ou des colonnes d'une matrice.

Exemple Par exemple :

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

--> A(2,:) = []
  A =
    1.    2.    3.
    7.    8.    9.
```

3.7 La commande find()

Si A est une matrice $n \times p$, dont les coefficients sont booléens (c'est-à-dire égaux à `%t` ou `%f`), on la transforme en un vecteur ligne de taille np en juxtaposant toutes ses lignes en une seule ligne. L'instruction `find(A)` donne alors, sous forme de vecteur ligne, la liste des coefficients de A qui ont la valeur `%t`.

Exemple Par exemple :

```
--> A = [ %t , %t , %f ; %f , %f , %f ; %t , %f , %t ]
  A =
    T T F
    F F F
    T F T

--> find(A)
 ans =
    1. 2. 7. 9.
```

Si A est une matrice de taille $n \times p$ dont les coefficients sont des nombres l'intruction $A == x$ renvoie une matrice de même taille dont les coefficients sont les réponses aux tests `A(i,j)==x`. On obtient donc une matrice dont les coefficients sont booléens.
Exemple Par exemple :

```
--> A = [ 1 , 1 , 0 ; 0 , 0 , 0 ; 1 , 0 , 1 ]
  A =
    1 1 0
    0 0 0
    1 0 1
--> A==1
 ans =
    T T F
    F F F
    T F T
```

On définit de même les matrices `A<x`, `A<=x` etc...

Application : l'instruction `length(find(A==x))` donne donc le nombre de coefficients de A qui sont égaux à x. On peut donc compter le nombre de coeffcients de A qui satisfont un test de comparaison.

4 Programmation en Scilab

Une succession de d'instructions est appelée un **script**. On peut les saisir directement dans la console Scilab, mais ceci a de nombreux inconvénients, car à chaque modification du script (même mineure), il faut le réécrire en entier. De plus, il est difficile de le sauvegarder pour une utilisation ultérieure. C'est pour ces raisons qu'on utilise systématiquement l'éditeur intégré Scinotes.

4.1 L'éditeur SciNotes

À partir de la fenêtre principale de Scilab, l'onglet Applications permet de lancer l'éditeur SciNotes.

Le fichier doit être sauvegardé avec le suffixe `.sce` pour indiquer au système que c'est un fichier Scilab. Pour assurer une compatibilité entre les divers systèmes d'exploitations, les

noms de fichiers doivent être en un seul mot (pas d'espace), sans accent, sans tiret et sans point (sauf le `.sce`). Par contre, on peut utiliser le caractère souligné _ pour un nom de fichier (il remplace le caractère espace).

Pour sauvegarder à partir de Scinotes, on utilise l'onglet Fichier puis Enregistrer sous si on veut donner un nouveau nom au fichier, ou Enregistrer si ce n'est pas nécessaire.

Le répertoire, appelé **répertoire courant**, dans lequel on travaille, est donné par la commande `pwd` ou via l'onglet Fichiers de la fenêtre principale. Il se change via la commande `chdir` ou via l'onglet Fichiers. Il est conseillé de créer un répertoire Scilab puis un sous-répertoire par séance de TP, pour sauvegarder à chaque fois tous les fichiers.

Pour exécuter un programme il faut le sauvegarder dans le répertoire courant, puis à partir de l'éditeur Scinotes, on clique sur l'onglet Exécuter, et on choisit Enregistrer et exécuter. Cette manipulation exécute le programme dans la console Scilab. Dans SciNotes, la touche Entrée ne sert donc plus à exécuter les commandes, mais simplement à passer à la ligne suivante (comme dans n'importe quel éditeur de texte).

Pour corriger/modifier un programme il suffit d'éditer le fichier avec SciNotes. Si le fichier a été fermé, ou peut l'ouvrir à nouveau avec l'onglet Fichier puis Ouvrir.

Un programme peut appeler un autre programme ou s'appeler lui-même. Lorsque l'exécution d'un programme ne s'arrête pas, on peut forcer l'arrêt en pressant simultanément es touches Ctrl et C. Si cela ne fonctionne pas, il ne reste qu'à fermer Scilab...

On peut ajouter des commentaires pour se rafraîchir la mémoire, ou pour expliquer le programme à une tierce personne. Il suffit de placer `//` en début de ligne : elle ne sera alors pas exécutée par Scilab. Cela permet aussi de chercher les erreurs : on empêche l'exécution de certaines lignes pour trouver celle(s) qui cause(nt) une erreur.

Les variables utilisées dans un programme sont **globales** : on peut les utiliser et les modifier aussi bien dans la console que dans un programme.

4.2 Affichage

Pour qu'un programme affiche du texte ou la valeur d'une variable, on utilise la commande `disp`.

Exemple Pour afficher le texte « My name is Brian » on utilise la commande `disp('My name is Brian')`.

Exemple Pour afficher la valeur de la variable `X`, on utilise la commande `disp(X)`.

⚠ la commande `disp('X')` n'affichera pas la valeur de la variable `X`, mais seulement le caractère `X`.

On peut afficher plusieurs objets sur la même ligne avec la commande `disp(objetn,....,objet2,objet1)`.

⚠ Il faut les mettre dans l'ordre inverse de celui souhaité pour l'affichage !

Dans certaines situations, on souhaite afficher un texte qui demande à l'utilisateur du programme de saisir une valeur numérique au clavier, valeur qu'on souhaite enregistrer dans une variable `X`.

Pour cela, on utilise la commande `X = input('Texte')`. Elle affiche la chaîne de caractère `Texte`, met le programme en pause, l'utilisateur saisie ensuite une valeur numérique au clavier, valide avec la touche Entrée, et Scilab enregistre cette valeur dans la variable `X`. Le programme reprend ensuite son exécution.

4.3 Structures conditionnelles

Scilab fournit trois possibilités de programmer l'alternative « Si…alors… ».

Version sans alternative

Si `test` est une variable booléenne, les commandes

```
début du programme

if test then

    instructions_if

end

suite du programme
```

ont deux exécutions possibles :

• une fois le début du programme exécuté, si `test` a la valeur `T` alors les instructions `instructions_if` sont exécutées et ensuite la suite du programme est exécutée ;

• si `test` a la valeur `F`, alors les instructions `instructions_if` sont ignorées, et c'est directement la suite du programme qui est exécutée.

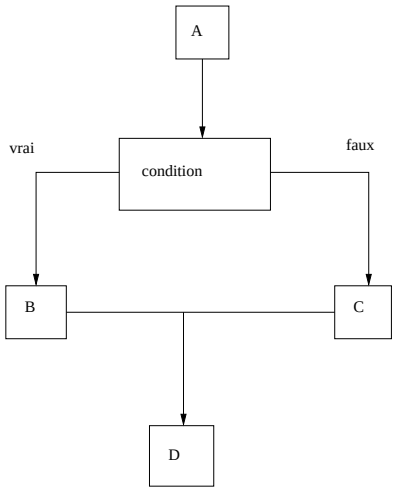
Version avec une alternative

Si `test` est une variable booléenne, les commandes

```
début du programme

if test then

    instructions_if

else

    instructions_else

end

suite du programme
```

ont deux exécutions possibles :
• une fois le début du programme exécuté, si `test` a la valeur `T` alors les instructions `instructions_if` sont exécutées (`instructions_else` sont ignorées) et ensuite la suite du programme est exécutée ;
• si `test` a la valeur `F`, alors ce sont les instructions `instructions_else` qui sont exécutées, puis la suite du programme (`instructions_if` sont ignorées).

C'est le cas le plus courant. On peut le visualiser sur le schéma suivant.



Version avec deux alternatives

Si `test1` et `test2` sont deux variables booléennes, les commandes

```
début du programme

if test1 then

    instructions_if

elseif test2 then

    instructions_elseif

else

    instructions_else

end

suite du programme
```

ont trois exécutions possibles :
• une fois le début du programme exécuté, si `test1` a la valeur `T` alors les instructions `instructions_if` sont exécutées (`instructions_elseif` et `instructions_else` sont ignorées) et ensuite la suite du programme est exécutée.

• si `test1` a la valeur `F` et si `test2` a la valeur `T`, alors ce sont les instructions `instructions_elseif` qui sont exécutées, puis la suite du programme (`instructions_if` et `instructions_else` sont ignorées).

• troisième possibilité : si `test1` a la valeur `F` et si `test2` a la valeur `F`, alors ce sont les instructions `instructions_else` qui sont exécutées, puis la suite du programme (`instructions_if` et `instructions_elseif` sont ignorées).

On peut généraliser à un nombre quelconque d'alternatives, en imbriquant plusieurs commandes `elseif`.

4.4 Boucles itératives

Boucle répétitive `for`
Cette commande s'emploie pour répéter une suite d'instructions un nombre fixé de fois, connu à l'avance. La syntaxe est la suivante :

```
début du programme

for i = m : h : M

    instructions_for

end

suite du programme
```

Le début du programme est exécuté, ensuite les instructions `instructions_for` sont répétées une première fois avec `i` prenant la valeur m, une seconde fois avec la valeur $m+h$, une troisième fois avec la valeur $m+2h$, ..., une dernière fois avec la valeur $m+kh$ où k est tel que $m+kh \leq M$ et $m+(k+1)h > M$. Ensuite Scilab continue avec la suite du programme.

Si $m : h : M$ n'est pas possible, les instructions `instructions_for` sont ignorées et Scilab passe directement à la suite du programme.

Lorsque $h = 1$ on eut omettre de le préciser : `for i = m : M`.

Boucle conditionnelle `while`

Par opposition à l'instruction `for`, l'instruction `while` s'emploie dès qu'on ne connaît pas à l'avance le nombre d'itérations à effectuer. La syntaxe est la suivante :

```
début du programme

while test

    instructions_while

end

suite du programme
```

où `test` est une variable booléenne.

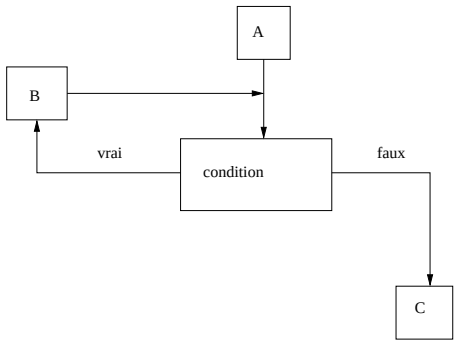
Le début du programme est exécuté. Ensuite tant que `test` garde la valeur `T`, les instructions `instructions_while` sont exécutées. Dès que `test` prend la valeur `F`, les instructions `instructions_while` sont ignorées, et Scilab passe à la suite du programme.

Les instructions `instructions_while` peuvent ne pas être exécutées du tout, si dès le départ `test` a la valeur `F`.

Si `test` a toujours la valeur `T`, alors le programme ne s'arrête jamais. La seule solution est de l'interrompre en pressant simultanément Ctrl et C.

⚠ Pour éviter ce genre d'erreurs, il faut que les instructions `instructions_while` modifie la valeur de `test`.

On a le schéma suivant :



4.5 Premier exemple de programme

Nous allons créer un programme permettant de calculer la factorielle d'un entier naturel n, c'est-à-dire le produit des entiers de 1 à n : $1 \times 2 \times 3 \times \cdots \times n$.

Analysons les calculs sur un exemple : le calcul de la factorielle de 7. On part de 1, on le multiplie par 2 (on obtient 2), puis le résultat par 3 (on obtient 6), puis on multiplie par 4 (24), par 5 (120), par 6 (820) et enfin par 7 ce qui donne 5040.

Essayons de généraliser : on a besoin d'une variable qui va stocker le résultat de chacune des multiplications, appelée `temp` (variable temporaire). À chaque itération on la multiplie par une variable `i` qui va prendre toutes les valeurs entières entre 2 et n. Initialement la variable `temp` a la valeur 1. À la fin des calculs `temp` a la valeur de la factorielle de n.

Puisqu'il faut répéter des opérations un nombre fixé à l'avance de fois, on va utiliser une boucle `for`.

On peut donc proposer le programme suivant :

```
n = input('donner l''entier naturel dont vous voulez
            calculer la factorielle : ');
// on demande à l'utilisateur une saisie au clavier
// qu'on affecte à une variable n
temp = 1; // on crée une variable temp et on lui affecte la valeur 1
for i = 1 : n     // boucle for
       temp = i * temp;    // on multiplie temp par i et on enregistre
                           // le résultat dans temp
end
disp(temp,' est :',n,'La factorielle de ')     // affichage du résultat
// remarquer que les espaces sont insérés dans la chaîne de caractères
```

On appuie sur F5 pour exécuter et sauver ce programme sous le nom `factorielle.sce`, dans le « répertoire courant ». Dans la console apparaît :

```
--> exec('/repertoire_courant/factorielle.sce', -1)
donner l'entier naturel dont vous voulez calculer la factorielle :
```

L'utilisateur saisie 7 et valide. On a alors l'affichage final :

```
-->exec('/repertoire_courant/factorielle.sce', -1)
donner l'entier naturel dont vous voulez calculer la factorielle :7

 La factorielle de

    7.

  est :

    5040.
```

On remarque sur cet exemple qu'il n'est pas très pratique d'utiliser plusieurs fois ce programme : on doit soit le lancer depuis SciNotes, soit depuis la console via la commande `--> exec('/repertoire_courant/factorielle.sce', -1)`. Pour simplifier cette utilisation, on préfère souvent programme des **fonctions**.

5 Fonctions en Scilab

5.1 Utilisation des fonctions en Scilab

Une fonction est un script (c'est-à-dire une suite d'instructions) à laquelle on donne un nom, afin de pouvoir l'utiliser dans différentes situations.

Cette suite d'instructions dépend de paramètres `x1`, `x2`, ..., `xm` qui sont spécifiés avant chaque utilisation : ce sont les **arguments** de la fonction. En sortie la fonction peut ne donner aucun résultat résultat numérique, n'en donner qu'un, ou encore en donner plusieurs.

On peut mettre une ou plusieurs fonction dans un même programme.

La syntaxe générale est la suivante (à écrire dans un fichier de SciNotes) :

```
function[y1,...,yn] = nom(x1,...,xm)

instructions

endfunction
```

`nom` correspond au nom qu'on choisit pour la fonction. `x1`, ..., `xm` sont ses arguments. `y1`, ..., `yn` sont ses résultats.

Pour une fonction sans argument la syntaxe serait :

```
function[y1,...,yn] = nom()

instructions

endfunction
```

et pour une fonction sans résultat numérique :

```
function[] = nom (x1,...,xm)

instructions

endfunction
```

Il faut ensuite utiliser la touche F5 ou l'onglet Exécuter pour »charger » la fonction dans Scilab. Elle s'utilise alors comme les fonctions prédéfinies (`cos`, `floor` etc ...) : dans la console, la commande `nom(x1,...,xm)`, où `x1`, ..., `xm` sont remplacés par des valeurs numériques, exécute la fonction `nom` avec les valeurs choisies pour les arguments.

⚠ Il faut sauvegarder dans le répertoire courant, sinon la fonction ne sera pas correctement « chargée » dans la console.

⚠ Le fichier peut porter le même nom que la fonction. Par contre la fonction ne peut pas porter le même nom qu'une variable.

Pour modifier/corriger une fonction on utilise SciNotes.

⚠ Il ne faut pas oublier de re - « charger » la fonction à chaque modification !

Les variables utilisées dans une fonction sont **locales** : elles sont effacées en fin d'exécution, et n'apparaissent pas dans le Navigateur de variables. Au contraire, les variables créées dans la console ou dans un programme sont appelées variables **globales** : on peut les utiliser et les modifier aussi bien dans la console que dans un programme.

⚠ Si `x` est une variable **globale**, on peut avoir dans une fonction une variable **locale** aussi appelée `x`. Cela ne crée pas de conflit, et la variable globale `x` n'est pas modifiée.

5.2 Un premier exemple de fonction

Reprenons l'exemple du calcul de la factorielle d'un entier naturel. Cette fois, on veut créer une fonction `fact` de telle sorte que la commande `fact(n)` exécutée dans la console donne en réponse la factorielle de `n`.

Dans ScinNotes on crée le fichier :

```
function[temp]=fact(n)
    temp=1;
    for i=1:n
        temp=temp*i;
    end
endfunction
```

Ensuite on le sauvegarde dans le répertoire courant sous le nom `fact.sce` et on « charge » la fonction dans la console avec la touche F5. On obtient :

```
-->exec('/repertoire_courant/fact.sce', -1)

-->
```

Pour calculer factorielle de 7 et de 12 on exécute les commandes `fact(7)` et `fact(12)` après l'invite `-->` :

```
--> fact(7)
 ans  =

    5040.

--> fact(12)
 ans  =

    4.790D+08
```

6 Les graphiques avec Scilab

6.1 Les fenêtres graphiques

En plus de la fenêtre principale et de l'éditeur SciNotes, il existe un troisième type de fenêtres : les fenêtres graphiques. Ces fenêtres sont nommées `Figure` suivi d'un numéro qui correspond à leur ordre de création.

On peut ouvrir plusieurs fenêtres graphiques simultanément. De plus, à l'intérieur d'une même fenêtre, on peut superposer des graphiques en les traçant les uns après les autres.

L'onglet Édition d'une fenêtre graphique permet de modifier le titre, d'ajouter une légende etc ...

Le même onglet permet aussi d'effacer la figure (sans fermer la fenêtre). Pour effacer simultanément tout le contenu de toutes les fenêtres graphiques, on exécute la commande `clf` dans la console.

6.2 Tracés de courbes

Si `x` et `y` sont deux vecteurs de même dimension, la commande `plot2d(x,y)` dessine une ligne brisée entre les points dont les abscisses sont données par le vecteur `x` et les ordonnées par le vecteur `y`.

Si `x` est un vecteur et `y` une matrice dont le nombre de colonnes est égal à la dimension de `x`, la commande `plot2d(x,y)` dessine des lignes brisées entre les points dont les abscisses sont données par le vecteur `x` et les ordonnées par chaque colonne du vecteur `y` (on obtient autant de lignes brisées que le nombre de colonnes de la matrice `y`).

Si `x` et `y` sont des matrices de même dimension, la commande la commande `plot2d(x,y)` dessine des lignes brisées entre les points dont les abscisses sont données par chaque colonne du vecteur `x` et les ordonnées par chaque colonne du vecteur `y` (on obtient autant de lignes brisées que le nombre de colonnes des matrices `x` et `y`).

On peut aussi utiliser l'instruction `plot` à la place de `plot2d`. Mais c'est une instruction adaptée du logiciel Matlab (logiciel payant dont Scilab est la version libre) qui est plus limitée : elle ne permet pas de tracer plusieurs courbes. Il est donc préférable d'utiliser `plot2d` qui est une instruction propre à Scilab.

Pour représenter la courbe d'une fonction numérique `f`, on utilise une ligne brisée dont les abscisses sont très proches, afin de donner l'illusion de courbe. Si on veut représenter `f` sur un intervalle $[a,b]$, on commence par créer le vecteur des abscisses `x` avec la commande `x = linspace(a,b,n)`, en prenant `n` suffisamment grand pour avoir l'illusion de courbe (mais sans faire exploser les temps de calcul !). Ensuite on crée le vecteur des ordonnées `y` avec la commande `y=f(x)`. Il ne reste plus qu'à exécuter la commande `plot2d(x,y)`.

⚠ Cela suppose que la fonction `f` a déjà été crée et chargée dans Scilab, et surtout qu'elle puisse s'appliquer coefficient par coefficient lorsqu'on utilise l'instruction `f(x)`. À cet effet, on pourra utiliser les opérations matricielles qui se font coefficient par coefficient : `.*`, `.^` et `./` etc ...

S'il n'est pas possible d'appliquer `f` à une matrice (ou un vecteur), on pourra utiliser une boucle `for`, mais le temps de calcul est plus long (tout en restant négligeable pour ce que nous ferons...). On pourra utiliser la syntaxe :

```
for i=1:n
     y(i) = f(x(i))
end
```

Il est possible de modifier les axes, l'échelle etc ...Pour cela, il suffit de se reporter à l'aide de la fonction `plot2d`. Il ne faut pas apprendre par coeur les différentes options (il y en a beaucoup trop !).

Pour tracer plusieurs courbes dans la même figure, on dispose de deux possibilités. La première est d'enchaîner à la suite les instructions `plot2d` :

```
plot2d(x,y1)
plot2d(x,y2)
plot2d(x,y3)
    .
    .
    .
```

Mais cette méthode a un gros défaut : l'échelle est recalculée par Scilab pour chaque courbe, mais sans retracer les courbes précédentes. Il est donc possible que seule la dernière courbe soit correcte, les autres étant été déformées.

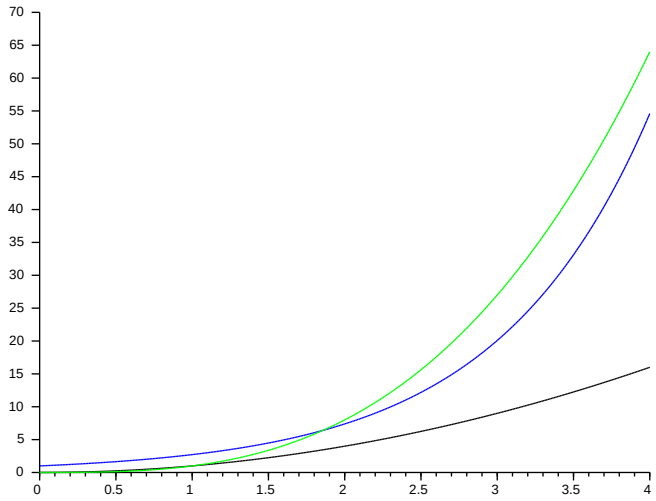
Il est donc préférable d'utiliser la syntaxe suivante : `plot2d(x',[y1',y2',y3',...])`.

⚠ Les ' sont mis pour avoir des vecteurs colonnes. Ils ne sont pas nécessaires si `x`, `y1`, `y2`, `y3`, ...sont déjà des vecteurs colonnes.

Exemple Les instructions :

```
--> x=linspace(0,4,501);

--> y1=x.^2; y2=exp(x); y3=x.^3;

--> plot2d(x',[y1',y2',y3'])
```

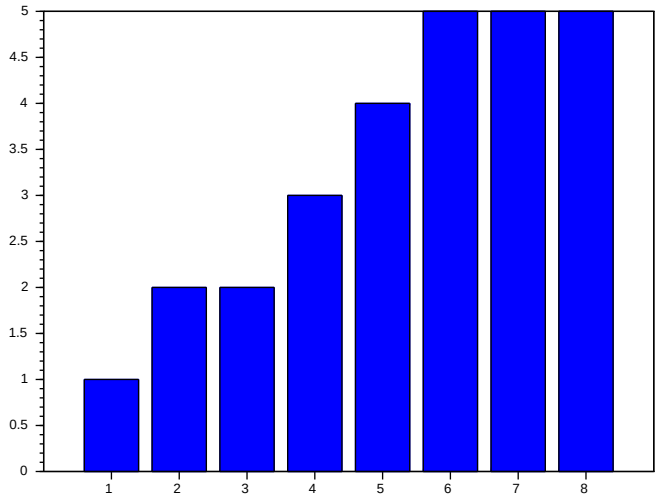
donnent la figure suivante :



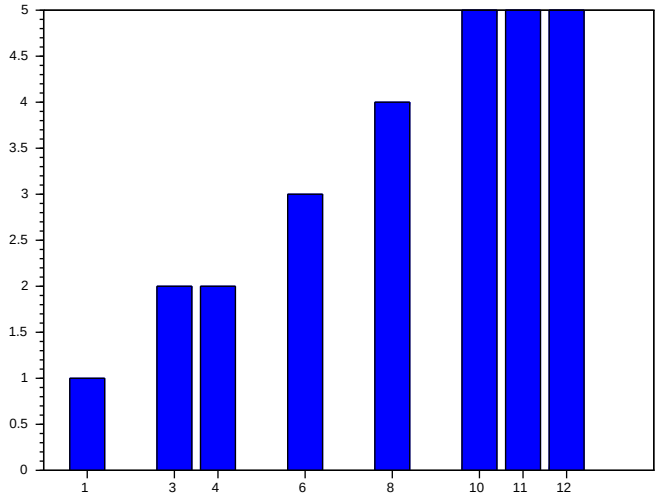
6.3 Tracés d'histogramme

Si `y` est un vecteur, la commande `bar(y)` trace un histogramme dont la hauteur de chaque barre est donnée par les coefficients de `y`. On peut aussi préciser les abscisses des barres en donnant un vecteur `x` de même longueur que `y` : `bar(x,y)`.

Exemple La commande `bar([1,2,2,3,4,5,5,5])` donne l'histogramme :



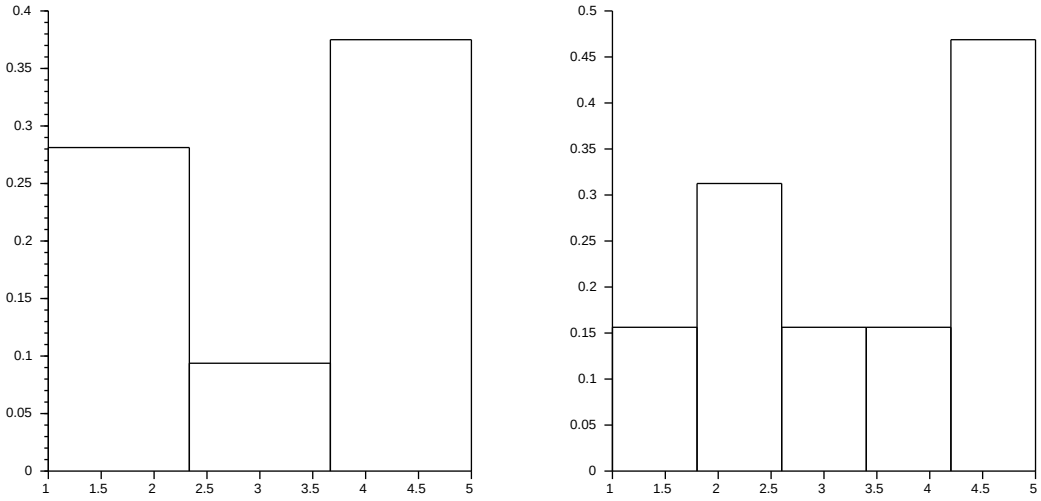
Exemple La commande `bar([1,3,4,6,8,10,11,12],[1,2,2,3,4,5,5,5])` donne l'histogramme :



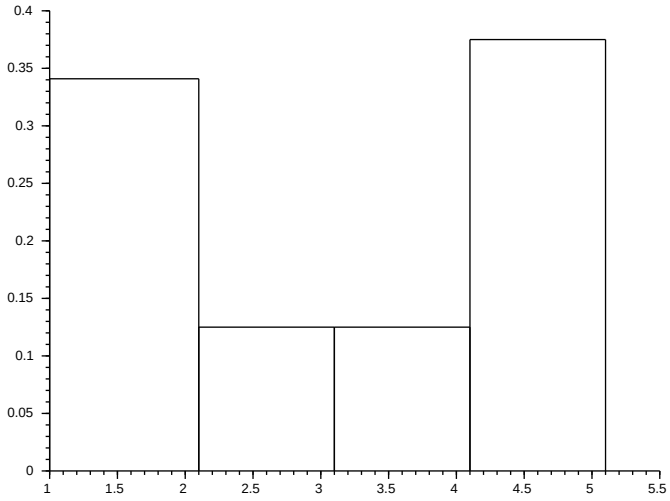
L' instruction `bar` suppose qu'on a déjà ordonnée les données par classe : `y` correspond aux effectifs et `x` à la liste des différentes classes.

Ce travail fastidieux peut être automatisé par l'instruction `histplot`. La commande `histplot(n,y)` réparti les coefficients de `y` en n classes équiréparties, et trace l'histogramme des effectifs normalisés (de telle sorte que l'aire totale soit égale à 1). On peut aussi préciser les classes via un vecteur ligne `c` :
`histplot(c,y)` trace un histogramme dont les classes sont délimitées par deux coefficients consécutifs de `c` : $[c(1),c(2)]$ puis $]c(k),c(k+1)]$.

Exemple Les commandes `histplot(3,[1,2,2,3,4,5,5,5])` et `histplot(5,[1,2,2,3,4,5,5,5])` donnent les histogrammes :



Exemple La commande `histplot([1,2.1,3.1,4.1,5.1],[1,2,2,3,4,5,5,5])` donne l'histogramme :



On voit sur cet exemple que le premier et le dernier rectangle ne sont pas de la même hauteur, alors qu'ils représentent le même effectif : cela est du au fait qu'ils n'ont pas la même largeur.

7 Simulation de lois de probabilités avec Scilab

7.1 La commande rand

Si r et s sont deux entiers naturels non nuls, la commande `A = rand(r,s)` crée une matrice `A` de dimension $r \times s$, chaque coefficient étant choisie selon la **loi uniforme sur** $[0,1]$, de manières « indépendantes ».

Si r et s sont deux entiers naturels non nuls, la commande `A = rand(r,s,'n')` crée une matrice `A` de dimension $r \times s$, chaque coefficient étant choisie selon la **loi normale centrée réduite**, de manières « indépendantes ».

Simulation d'une loi uniforme discrète. Pour simuler une réalisation de la loi $\mathscr{U}(\llbracket a,b \rrbracket)$, à partir de l'instruction `rand()`, on peut procéder ainsi :

```
subdi=linspace(0,1,b-a+2);  // découpe de l'intervalle [0,1] en
   // (b-a+1) morceaux de même longueur
alea=rand();
for k=1:(b-a+1)
      if subdi(k)<= alea & alea<subdi(k+1) then x=k; end
end
disp(x)
```

En effet si on découpe l'intervalle $[0,1]$ en n morceaux, la probabilité que `rand()` donne une valeur dans un morceau donné est égale à $\dfrac{1}{n}$ (voir cours de maths sur les lois à densité).

Avec l'instruction `find` on peut ne pas utiliser de boucle `for` :

```
subdi=linspace(0,1,b-a+2);  // découpe de l'intervalle [0,1] en
    // (b-a+1) morceaux de même longueur
alea=rand();
x=find( subdi <= alea & alea < ( subdi+1/(b-a+1) ) );
disp(x)
```

En effet, on recherche le seul sous-intervalle de la subdivision qui contient la valeur aléatoire `rand()`.

Simulation d'une loi binomiale. Pour simuler une réalisation de la loi $\mathscr{B}(n,p)$, à partir de l'instruction `rand()`, on peut procéder ainsi :

```
x=0;
for k=1:n
   if rand()<p then x=x+1; end
end
disp(x)
```

En effet on compte ainsi le nombre de succés lors de n répétitions d'une expérience qui donne succés avec probabilité p.

Avec l'instruction `find` on peut ne pas utiliser de boucle `for` :

```
alea=rand(1,n);
x=length(find( alea<p ));
disp(x)
```

En effet, on compte le nombre de coefficients du vecteur ligne `alea` qui sont strictement inférieurs à p.

Simulation d'une loi géométrique. Pour simuler une réalisation de la loi $\mathscr{G}(p)$, à partir de l'instruction `rand()`, on peut procéder ainsi :

```
x=0;
while rand()>=p
   x=x+1;
end
disp(x)
```

Simulation d'une loi de Poisson. Pour simuler une réalisation de la loi $\mathscr{P}(\lambda)$, à partir de l'instruction `rand()`, on peut procéder ainsi :

```
x=0;
p=exp(-lambda);
s=p;
alea=rand();
while s<alea
   x=x+1;
   p=p*lambda/x;
   s=s+p;
end
disp(x)
```

Remarquer qu'on calcule les fractions $\dfrac{\lambda^k}{k!}$ par récurrence. En effet, les nombres λ^k et $k!$ sont souvent trop grands pour le logiciel.

L'idée est que la probabilité que `rand()` donne une valeur située dans l'intervalle $\left[\mathrm{e}^{-\lambda}\sum_{j=0}^{n-1}\dfrac{\lambda^j}{j!}, \mathrm{e}^{-\lambda}\sum_{j=0}^{n}\dfrac{\lambda^j}{j!}\right[$ est égale à $\mathrm{e}^{-\lambda}\dfrac{\lambda^n}{n!}$.

7.2 La commande grand

Variables discrètes

Si r et s sont deux entiers naturels non nuls, la commande `A = grand(r,s,'uin',1,n)` crée une matrice `A` de dimension $r \times s$, chaque coefficient étant choisie selon la **loi uniforme sur** $[\![1, n]\!]$, de manières « indépendantes ».

Si r et s sont deux entiers naturels non nuls, la commande `A = grand(r,s,'bin',n,p)` crée une matrice `A` de dimension $r \times s$, chaque coefficient étant choisie selon la **loi binomiale de paramètres** `(n,p)`, de manières « indépendantes ».

Si r et s sont deux entiers naturels non nuls, la commande `A = grand(r,s,'geom',psucces)` crée une matrice `A` de dimension $r \times s$, chaque coefficient étant choisie selon la **loi géométrique de paramètre** `psucces`, de manières « indépendantes ».

Si r et s sont deux entiers naturels non nuls, la commande `A = grand(r,s,'poi',lambda)` crée une matrice `A` de dimension $r \times s$, chaque coefficient étant choisie selon la **loi de Poisson de paramètre** `lambda`, de manières « indépendantes ».

Variables à densité

Si r et s sont deux entiers naturels non nuls, la commande `A = grand(r,s,'unf',a,b)` crée une matrice `A` de dimension $r \times s$, chaque coefficient étant choisie selon la **loi uniforme sur** $[a, b]$, de manières « indépendantes ».

Si r et s sont deux entiers naturels non nuls, la commande `A = grand(r,s,'exp',mu)` crée une matrice `A` de dimension $r \times s$, chaque coefficient étant choisie selon la **loi exponentielle de paramètre** `1/mu`, de manières « indépendantes ».

⚠ Le paramètre à indiquer pour $\mathscr{E}(\lambda)$ n'est donc λ mais l'espérance $\mu = \dfrac{1}{\lambda}$.

Si r et s sont deux entiers naturels non nuls, la commande `A = grand(r,s,'nor',mu,sigma)` crée une matrice `A` de dimension $r \times s$, chaque coefficient étant choisie selon la **loi de normale de paramètres** `mu` **et** `sigma^2`, de manières « indépendantes ».

⚠ Les paramètres sont la moyenne et **l'écart-type**, alors que dans le cours de mathématiques il est d'usage de donner la moyenne et la **variance**.