Chapitre 10 : Introduction à Scilab

Scilab (qui est une contraction de *scientific laboratory*) est un logiciel libre de calcul numérique multiplateforme fournissant un environnement de calcul et de développement pour des applications scientifiques. Scilab a été développé en langage Fortran à l'INRIA (Institut National de Recherche en Informatique et en Automatique) à partir de 1982, et offre sensiblement les mêmes possibilités que le logiciel (payant) Matlab. Vous pourrez télécharger la dernière version à l'url `https://www.scilab.org/fr/download/5.5.1`.

Vis-à-vis de ce qui sera étudié en classe préparatoire, Scilab offre sensiblement les mêmes possibilités que Python, à l'exception de la gestion des bases de données. Mais il est important de retenir que si Python est un langage de progammation complet, qui permet via certaines bibliothèques (comme `math`) de faire du calcul scientifique ou de tracer des graphes, Scilab est au contraire un simple logiciel de calcul scientifique, qui offre (pour cette raison) les fonctionnalités élémentaires d'un langage de programmation. En pratique, Python offre donc beaucoup plus de possibilités.

Comme sous Pyzo, un éditeur de texte est disponible, afin d'écrire des codes plus longs. Son nom est *scinotes*, et on y accède en cliquant sur l'onglet "Application" puis sur "scinotes", ou en entrant directement `scinotes` dans la console. Une fois un code écrit dans l'éditeur de texte, on demande à Scilab de le *compiler* en cliquant sur l'onglet "Exécuter", puis en cliquant sur "Enregistrer et exécuter" (plus simplement, il suffit d'appuyer sur la touche F5). Cette action demande à Scilab de compiler le code tapé, c'est-à-dire de vérifier qu'il n'y a pas d'erreur de syntaxe. Si c'est le cas, Scilab met alors le code en mémoire, qui pourra ensuite être utilisé dans la fenêtre principale ; sinon vous verrez apparaître un message d'erreur dans la console.

1 Fonctionnalités élémentaires

Les opérations arithmétiques élémentaires et l'affectation s'appellent de la même manière en Scilab et en Python, et leur comportement est le même.

```
-->1+3
 ans  =
    4.

-->3*12
 ans  =
    36.
```

```
-->2**4
 ans  =
    16.

-->x+1
  !--error 4
Variable non définie : x
```

```
-->x=2
 x  =
    2.

-->x+1
 ans  =
    3.
```

Scilab connait les constantes mathématiques usuelles. On obtient généralement celles-ci en utilisant le symbole `%`, puis le nom de la constante considérée. Ainsi, le nombre π est obtenu en entrant `%pi`, le nombre complexe i est obtenu en entrant `%i`, et le réel e est associé à `%e`.

D'autre part, il faut remarquer que Scilab n'est pas programmé pour effectuer des calculs exacts : en particulier, si on entre la fraction $\frac{1}{3}$, Scilab renvoie la valeur 0.3333333. Cet exemple laisse à penser que Scilab travaille avec des valeurs approchées exactes à 10^{-7} près (ou du moins *renvoie* des valeurs approchées à 10^{-7} près). Mais la réalité est plus complexe : la valeur approchée de la fraction $\frac{100}{3}$ donnée par Scilab est 33.333333. En fait, Scilab renverra des expressions qui font apparaitre au plus 10 caractères (le signe et le point comptant pour un caractère). Donc on a *a priori* une précision de 10^{-7} sur les réels dans $[0, 10[$, une précision de 10^{-6} sur les réels dans $[10, 99[$, une précision de 10^{-5} sur les réels dans $[100, 999[\ldots$ Par ailleurs, la manipulation des flottants fait apparaître les mêmes problématiques que celles vues sous Python ; les flottants ont en fait une précision maximale de 16 chiffres décimaux, et leur valeur absolue est comprise entre 10^{-308} et 10^{308}.

```
-->1/3
 ans  =
    0.3333333

-->1.1234567
 ans  =
    1.1234567
```

```
-->1.12345678
 ans  =
    1.1234568

-->12345678
 ans  =
    12345678.
```

```
-->123456789
 ans  =
    1.235D+08
```

Le nombre de caractères affichés lors du renvoi d'un flottant peut être modifié grâce à la commande `format(n)`, n étant le nombre de caractères que l'on souhaite voir Scilab afficher, qui doit nécessairement

appartenir à l'ensemble $[\![2, 25]\!]$.

```
-->format(5)
-->%pi
 %pi  =
    3.14
```

```
-->format(7)
-->%pi
 %pi  =
    3.1416
```

Pour finir, voilà deux considérations d'ordre général :

Remarque 1.1. — vous avez dû remarquer que le symbole `ans` apparaissait souvent ; il permet en fait d'employer la dernière valeur calculée par Scilab sans avoir besoin de taper celle-ci au clavier.

— le logiciel est fourni avec une rubrique « aide », qui vous permettra de trouver tout ce dont vous aurez besoin. Pour ouvrir celle-ci, cliquer sur l'icône figurant un point d'interrogation. Vous aurez ensuite accès à la table des matières de l'aide, mais vous pourrez également effectuer une recherche par mots-clés.

2 Définition de fonctions

Contrairement à Python, le langage de Scilab ne contient pas de notion d'indentation ; les *blocs d'instructions* sont délimités par des mots clés appelés *délimiteurs*. Si le délimiteur ouvrant un bloc d'instructions correspond à l'instruction elle-même, le délimiteur fermant un bloc est lui de la forme `end`, et n'aura d'autre rôle que de signaler à la machine que l'on est arrivé au bout du bloc.

Vous remarquerez néanmoins que l'éditeur de texte effectue de lui-même des indentations lorsque vous programmez une fonction. Celles-ci n'ont aucun rôle syntaxique (c'est-à-dire que vous pouvez très bien les effacer), mais permettent d'obtenir des programmes plus simples à relire ou débogger. Il est donc conseillé d'indenter ses blocs d'instructions exactement comme on le fait en Python.

Traditionnellement, on entre le code d'une fonction dans l'éditeur de texte. Celui-ci fournit une « aide à la programmation » autre que l'indentation, censée vous éviter des erreurs de syntaxe, mais attention celle-ci peut également induire en erreur.

Pour définir une fonction appelée par exemple f, qui attend n arguments $x_1, \dots, x_n$ et qui est censée renvoyer p valeurs $y_1, \dots, y_p$, on utilise la syntaxe suivante :

```
function [y1,...,yp]=f(x1,...,xn)
        //Calcul de y1,...,yp grace a x1,...,xn
endfunction
```

On peut d'ores et déjà remarquer deux choses :

— l'instruction `endfunction` est un délimiteur, qui sert uniquement à faire comprendre à Scilab que la définition de la fonction est terminée ;

— il n'y a pas d'instruction du type `return(y1,...,yp)`, car la ligne `function [y1,...,yp]= f(x1,...,xn)` fait déjà comprendre à Scilab que ce sont les variables `y1,...,yp` qu'il faudra renvoyer à la fin ;

— les commentaires se font derrière les symboles `//`.

Exemple 2.1. Écrire, en version Python et en version Scilab une fonction renvoyant x^2 à partir d'un flottant x.

Remarque 2.2.
On prendra garde au fait que les codes suivants ne sont pas valables.

```
function [y]= f(x)
    f(x)=x**2
endfunction
```

```
function [y]= f(x)=x**2
endfunction
```

Bien évidemment, Scilab connait les fonctions usuelles :

— `abs` est la fonction valeur absolue ;
— `sqrt` est la fonction racine carré (sqrt est l'abréviation de l'anglais SQuare RooT) ;
— `log`, `exp` sont les fonctions logarithme népérien (c'est la notation anglaise) et exponentielle ;
— `acos`, `asin`, `atan` désignent respectivement les fonctions arccosinus, arcsinus, arctangente.

3 Conditions booléennes et branchements conditionnels

Les conditions booléenes se fabriquent et se manipulent comme en Python, mais il y a quelques différences de syntaxe à noter : le symbole $\neq$ s'obtient en entrant `<>`, le vrai et le faux s'appellent avec les commandes `%t` et `%f` (car ce sont des constantes), et les connecteurs logiques sont obtenus avec `&,|,~`.

==	égal à
<>	différent de
<	inférieur à
<=	inférieur ou égal à
>	supérieur à
>=	supérieur ou égal à

$\%t$	true
$\%f$	false
&	et
|	ou
$\sim$	non

Attention, si le vrai et le faux s'appellent avec `%t` et `%f`, Scilab renvoie lui les lettres `T` et `F` pour s'y référer.

```
-->2>3 | ~1==0 | %f
 ans  =
  T
```

Les primitives `&` et `|` peuvent être remplacées par `and` et `or`, mais dans ce dernier cas la syntaxe est différente : le code `A & B` peut en fait être écrit `and([A,B])` (et de même pour le « ou »).

Le branchement conditionnel est implémenté par la structure suivante.

```
if Condition(1) then Action(1)
elseif Condition(2) then Action(2)
else Action(3)
end
```

Remarques 3.1.

— l'instruction `end` est un délimiteur de fin de bloc : il est donc indispensable, sinon Scilab renverra un message d'erreur à la compilation.
— attention à la différence de syntaxe : on écrit `elseif` en Scilab et `elif` en Python.

Exercice 3.2. Écrire un code Scilab qui implémente la fonction :

$$f : x \mapsto \begin{cases} x \ln x & \text{si } x > 0 \\ 0 & \text{sinon.} \end{cases}$$

4 Listes

Une liste est pour Scilab une suite d'éléments séparés par des virgules, cette suite étant délimitée par des crochets. Cette notion de liste est analogue à celle existant sous Python, mais nous allons voir qu'il existe de nombreuses différences syntaxiques entre les fonctions des deux langages s'appliquant aux listes (en fait toutes les primitives sont différentes) :

— `length` calcule la longueur d'une liste.

— La liste vide est définie par `[ ]`.

— Les éléments d'une liste L de longueur n sont indéxés de 1 à n (contrairement à Python...), et on accède au i-ème élément avec la commande `L(i)`. Cet accès peut s'effectuer en lecture et en écriture.

— Pour concaténer deux listes `L1` et `L2`, on utilise la commande `[L1,L2]`.

— L'égalité et la différence de deux listes ne se testent pas aussi aisément qu'en Python : les commandes `==` et `<>` effectuent des tests *coordonnée par coordonnée*. Néanmoins, on peut tester directement l'égalité entre deux listes en utilisant la primitive `and`.

```
-->l1=[1,2,3]
 l1  =
    1.    2.    3.

-->l2=[1,2,4]
 l2  =
    1.    2.    4.

-->l1==l2
 ans  =
  T T F

-->and(l1==l2)
 ans  =
  F
```

— Contrairement à Python, le symbole + ne permet pas de concaténer deux listes, mais de faire la somme coordonnée par coordonnée de deux listes. Cela provient du fait que pour Scilab, une liste est un vecteur. De même on peut faire la différence coordonnée par coordonnée de deux vecteurs en utilisant `-`, alors que le produit, le quotient et les puissances s'obtiennent respectivement avec `.*`, `./` et `**` (attention, on n'ajoute pas de point pour la fonction puissance).

```
-->V1=[1,2,3]
 V1  =
    1.    2.    3.

-->V2=[4 5 6]
 V2  =
    4.    5.    6.

-->V1+V2
 ans  =
    5.    7.    9.

-->V1./V2
 ans  =
    0.25    0.4    0.5
```

Plus généralement, les fonctions usuelles s'appliquent coordonnée par coordonnée aux listes.

```
-->sqrt([1 2 3 4])
 ans  =
    1.    1.4142136    1.7320508    2.

-->cos([0, %pi/2, %pi])
 ans  =
    1.    6.123D-17  - 1.
```

— Il n'y a pas en Scilab de primitive permettant de tester l'appartenance d'un objet à un vecteur, comme le fait `in` sous Python. En revanche, on peut obtenir le même résultat en exploitant le fait que les fonctions s'appliquent coordonnée par coordonnée aux vecteurs : la commande `or(x==L)` teste si l'objet `x` appartient à la liste `L` ou pas.

```
-->x=1
 x  =
    1.
-->y=0
 y  =
    0.

-->L=[1,2,3]
 L  =
    1.    2.    3.

-->or(x==L)
 ans  =
  T
-->or(y==L)
 ans  =
  F
```

— On a parfois besoin d'entrer des vecteurs de grande taille, et on peut pour cela utiliser certains raccourcis. La commande `V= ones(1,n)` affecte à la variable `V` la valeur suivante : la liste de taille n dont chaque coordonnée est égale à 1.
De manière analogue, la commande `V= zeros(1,n)` affecte à la variable `V` la valeur suivante : la liste de taille n dont chaque coordonnée est égale à 0. La présence d'un paramètre égal à 1 dans ces deux lignes de code sera expliquée lorsque nous introduirons les matrices.

— Soient $x < y$ des réels et $a > 0$. La commande `[x: a : y ]` renvoie la liste formée des termes inférieurs à y de la suite arithmétique de premier terme x et de raison a. Cette primitive nous permettra plus tard de tracer des courbes représentatives de fonctions.

```
-->[1:1:4]
 ans  =
    1.    2.    3.    4.

-->[0:0.1:0.55]
 ans  =
    0.  0.1  0.2  0.3  0.4  0.5
```

— Scilab ne permet pas l'affectation parallèle.

Donnons encore un exemple soulignant un point qui pose fréquemment des problèmes lorsque l'on implémente une fonction qui renvoie deux objets : observez bien comment faire pour récupérer en pratique ces deux objets.

```
function [a,b]=f(x,y)
    a=x+y
    b=x*y
endfunction
```

```
-->f(1,1)
 ans  =
    2.
```

```
-->[a,b]=f(1,1)
 b  =
    1.
 a  =
    2.
```

Exercice 4.1.
Programmer une fonction qui prend en entrée une liste x de longueur n, puis renvoie une liste y de longueur n telle que $\forall i \in [\![1, n]\!],\ y(i) = x(i) + 1$.

5 Les chaînes de caractères

Les chaînes de caractères sont définies comme en Python, à l'aide d'une suite de caractères mise entre guillemets ango-saxons ou apostrophes. Mais attention, il s'agit d'une structure assez pauvre en Scilab, car il n'y a pas d'accès possible aux termes d'une chaîne de caractère, que ce soit en lecture ou en écriture.
On peut bien entendu calculer la longueur d'une chaîne en utilisant la primitive `length`, et concaténer deux chaînes grâce au symbole `+` (comme en Python, contrairement aux listes en Scilab).
Dans le cas où la chaîne que l'on veut construire contient elle-même des apostrophes, on doit doubler celles-ci pour éviter tout message d'erreur.

```
-->x='j'aime bien Scilab'
        !--error 276
Opérateur, virgule ou point-virgule manquant.
```

```
-->x='j''aime bien Scilab'
 x  =
 j'aime bien Scilab
```

Remarques 5.1.

— Scilab ne possède pas de notion de tuple.

— Sous Scilab, les listes doivent être homogènes, c'est-à-dire qu'une liste ne peut contenir que des objets de même type.

6 Tracé de graphes

6 A) La primitive graphique plot2d

Pour tracer la courbe représentative d'une fonction f sur un intervalle $[a, b]$, nous allons fabriquer une liste $[x_1, \ldots, x_n]$ contenant un certain nombre de réels de cet intervalle, et ensuite fabriquer la liste image $[f(x_1), \ldots, f(x_n)]$ associée. Le logiciel se chargera alors de placer sur un graphe chacun des points M_i de coordonnés $(x_i, f(x_i))$, puis de relier chaque point M_i à son successeur M_{i+1} par des segments de droite. On obtient ainsi une approximation du graphe de f par une ligne polygonale. Tous les logiciels de calcul scientifique fonctionnent de cette manière.
Le code générique commandant le tracé d'une fonction f est le suivant :

```
-->x=[x1,...,xn]
-->y=f(x)
-->plot2d(x',y')
```

Le vecteur `x` contient donc les abscisses des points M_i décrits ci-dessus, et le vecteur `y` leurs ordonnées.

On construit d'abord le vecteur x des points à partir desquels on souhaite approcher le graphe (de leur nombre dépendra la précision de ce dernier), puis on applique f à partir de celui-ci. On utilise enfin la primitive `plot2d` pour relier les points du plan entre eux. En réponse, Scilab ouvre alors une fenêtre donnant le graphe demandé.

Exemple 6.1.
Pour obtenir la courbe représentative de la fonction cosinus sur $[0, 2\pi]$ ci-dessous, on a pris comme vecteurs d'abscisses `x=[0:0.1:2*%pi]` : on utilise donc à peine 62 points.

```
-->x=[0:0.1:2*%pi];

-->y=cos(x);

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

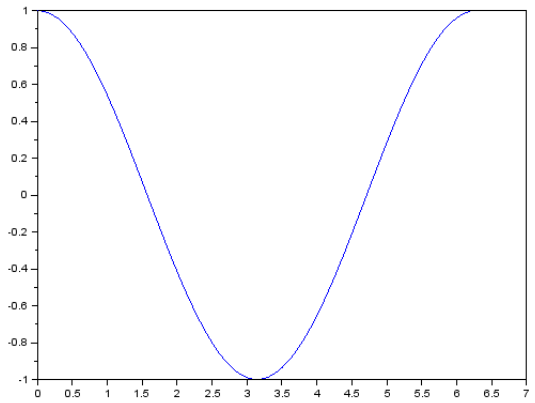

Figure 6.2 - Graphe de la fonction cosinus sur $[0, 2\pi]$.

Remarques 6.3.

— Si vous entrez une ligne de code dans la console et que vous terminez celle-ci par `;` alors Scilab n'affichera pas la réponse à votre code. Cela est assez pratique lorsque l'on crée des vecteurs de grande taille, afin de ne pas saturer l'écran de la console.

— Pourquoi employer le symbole $'$ après x et y ? Car ce symbole correspond à la transposition matricielle, et le logiciel est programmé pour tracer des colonnes et pas des lignes !

6 B) Quelques options de plot2d

On peut faire interagir différentes options avec la primitive `plot2d`. On entre celles-ci comme des arguments supplémentaires de la fonction :

```
plot2d(x',y',option1,option2,....)
```

Tracé de deux courbes sur un même graphique

Tant que la fenêtre graphique n'est pas fermée, Scilab rajoutera automatiquement toute nouvelle courbe sur le même graphe.

Mais on peut également tracer plusieurs graphes simultanément, et Scilab tracera alors ces deux graphes avec des couleurs différentes. On peut également choisir la couleur donnée à chaque graphe, en utilisant l'option `style=[...]`, et en faisant figurer entre les crochets les numéros associés aux couleurs choisies. Cela donne par exemple :

```
plot2d([listex' listex'],[f(listex)' g(listex)'],style=[1 2])
```

qui tracera la courbe représentative de f en noir et celle de g en bleu, sur l'intervalle associé à x. Voilà enfin le numéro associé à quelques couleurs :

1	noir	5	rouge vif
2	bleu	6	mauve
3	vert clair	29	rose
4	cyan	32	jaune orangé

Ajout d'une légende et d'un titre

Lorsque plusieurs courbes sont tracées sur un même graphique, il est nécessaire d'ajouter une légende afin de pouvoir déterminer à quelle fonction correspond chacun des tracés. On ajoute pour cela l'option suivante en argument de la primitive `plot2d`, dans le cas où l'on a tracé n fonctions $f1, f2, \ldots, fn$: `leg='f1@f2@...@fn'`. On utilise dans cette option des guillemets pour spécifier au logiciel que la légende est en fait composée de chaînes de caractères.

Pour donner un titre au graphique obtenu, on entre simplement le code `xtitle('Mon titre')` après avoir tracé le graphique.

Enfin, pour nommer les axes de coordonnées, on emploiera les codes `xlabel('Titre abscisse')` et `ylabel('Titre ordonnée')`.

Voilà enfin un exemple qui utilise tout ce qui vient d'être vu. Notez au passage l'emplacement des symboles primes : on trace `[listey1' listey2']`, et pas `[listey1 listey2]'`.

```
-->listex=[0.01:0.05:2];

-->listey1=log(listex);

-->listey2=exp(listex);

-->plot2d([listex' listex'],
    [listey1' listey2'],
    style=[2 3],leg="ln@exp")

-->xtitle("Le logarithme et
    l''exponentielle");
```

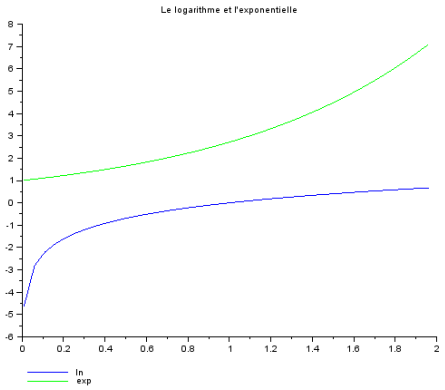

Figure 6.4 - Courbes représentatives de l'exponentielle et du logarithme sur $[0,01;2]$.

On observe que la courbe du logarithme n'est pas parfaitement lisse : cela est dû au fait que l'on n'a pas choisi suffisamment de points dans l'intervalle $[0,01;2]$ pour tracer notre graphe.

Placement de points sur un graphe

Comme nous l'avons vu, à chaque fois que Scilab trace un graphe, il se contente de placer les points qu'on lui donne puis de les relier par des segments de droite.

Pour se contenter de tracer les points sans les relier, il suffit d'employer une option de style : la plus couramment utilisée est le numéro -2, avec lequel chaque point sera représenté par une croix $\times$.

7 Structures répétitive et itérative

7 A) Boucle while

Il n'y a aucune différence fondamentale entre les boucles while sous Python ou Scilab. La seule chose qui change est la syntaxe : on a en Scilab un délimiteur `do` pour ouvrir le bloc, et un `end` pour le fermer. Précisment, la syntaxe est la suivante :

```
while condition do
    Action
end
```

Par exemple, les fonctions ci-dessous permettent respectivement de calculer la partie entière d'un réel positif, et la somme des n premiers entiers.

```
function [n]=partie_entiere(x)
    n=0
    while n<=x do
        n=n+1
    end
    n=n-1
endfunction
```

```
function [S]=somme(n)
    S=0
    k=0
    while k<=n do
        S=S+k,
        k=k+1
    end
endfunction
```

Remarque 7.1.
Dans la déclaration de `somme`, les lignes de code `S=S+k` et `k=k+1` sont séparées par une virgule : lorsque l'on effectue plusieurs opérations à la suite à l'intérieur d'un même bloc, on prendra l'habitude de les séparer par des virgules (sinon des bugs peuvent apparaître).

7 B) Boucle for

La boucle for possède une syntaxe plus explicite en Scilab qu'en Python, mais celle-ci est moins expressive puisqu'on ne peut demander à une variable de parcourir les coordonnées d'une liste (on est obligé de parcourir les indices).

Sous Scilab, la syntaxe qui permet de répéter une action `Action(k)` pour k variant de a jusqu'à b est la suivante.

```
for k=a:b do
        Action(k)
end
```

Par exemple, les fonctions ci-dessous permettent respectivement de calculer la somme des n premiers entiers, et la liste `[0,...,n]`.

```
function [S]=somme(n)
    S=0
    for i=1:n do
        S=S+i
    end
endfunction
```

```
function [L]=construction_liste(n)
    L=[]
    for k=0:n do
        L=[L,k]
    end
endfunction
```

Exercice 7.2.
Écrire un programme qui prend en entrée un vecteur, puis renvoie l'indice du premier terme nul de celui-ci (on utilisera d'abord une boucle while, puis une boucle for).

Exercice 7.3.
Écrire une procédure `dichotomie` qui prend en entrée un réel $\varepsilon > 0$, une fonction f et deux réels $a \leqslant b$ tels que $f(a) \leqslant 0$ et $f(b) \geqslant 0$, puis renvoie une valeur approchée à ε près d'un réel $c \in [a, b]$ tel que $f(c) = 0$.

8 Les matrices sous Scilab

8 A) Qu'est-ce qu'une matrice pour Scilab ?

Dans le langage de Scilab une matrice est une liste dont chaque coordonnée est une suite de nombres séparés par des espaces, la $i^{\text{ième}}$ coordonnée correspondant à la $i^{\text{ième}}$ ligne de la matrice (on remplit donc les matrices ligne par ligne). Mais on est obligé de séparer les coordonnées de la liste par des points virgules : sinon Scilab croira que l'on entre simplement une liste de nombres.

```
-->M=[1 1 1;2 4 8;3 9 27]
 M  =
    1.    1.    1.
    2.    4.    8.
    3.    9.    27.
```

8 B) Primitives de lecture et de construction

— La primitive `size` renvoie la dimension de la matrice qu'on lui donne en argument, sous la forme d'une liste : (nombre de lignes, nombre de colonnes).

— La commande `zeros(n,p)` (resp. `ones(n,p)`) permet de créer la matrice de taille (n, p) dont tous les coefficients sont égaux à 0 (resp. 1).

— Étant donnée une matrice `M`, on accède à son terme d'indice (i, j) avec la commande `M(i,j)`.

— On peut construire une matrice en en juxtaposant plusieurs. Considérons par exemple la matrice suivante, dont on donne une représentation par blocs :

$$M = \left(\begin{array}{c|cc} 1 & 2 & 3 \\ \hline 4 & 5 & 6 \\ 7 & 8 & 9 \end{array}\right) = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$$

Chacun des blocs A, B, C et D est alors appelé « une sous-matrice de la matrice M. » Les lignes de code suivantes indiquent comment entrer M par blocs sous Scilab.

```
-->A=[1]
 A  =
    1.

-->B=[2  3]
 B  =
    2.    3.

-->C=[4; 7]
 C  =
    4.
    7.
```

```
-->D=[5 6 ; 8 9]
 D  =
    5.    6.
    8.    9.

-->M=[A B; C D]
 M  =
    1.    2.    3.
    4.    5.    6.
    7.    8.    9.
```

— *A contrario*, on peut extraire la i-ième ligne (resp. colonne) d'une matrice `M`, en employant la commande `M(i,:)` (resp. `M(:,i)`).

— Plus généralement, si `A` est une matrice de taille $n \times p$, si $V = [i_1 \ \ldots \ i_{n'}]$ (resp. $W = [j_1 \ \ldots \ j_{p'}]$) est une liste d'entiers compris entre 1 et n (resp. compris entre 1 et p), alors la commande `A(V,W)` désigne la matrice de taille $n' \times p'$ formée par l'intersection des lignes $i_1, \ldots, i_{n'}$ et des colonnes $j_1, \ldots, j_{p'}$ de A.

8 C) Opérations sur les matrices

— Les fonctions usuelles s'appliquent *coordonnée par coordonnée* aux matrices, comme elles le faisaient déjà avec les listes.

— La commande `M'` permet de calculer la transposée d'une matrice `M`.

— Pour multiplier une matrice par un nombre complexe, il suffit d'employer la commande $*$: chaque coordonnée de la matrice sera alors multipliée par le scalaire considéré. Scilab connait en outre les opérations algébriques élémentaires sur les matrices, et il suffit pour les appeler d'employer les symboles usuels : $+, -, *$ et $**$.

— Si `M` est inversible, Scilab interprétera la commande `1/M` en calculant l'inverse de `M`. De même, la commande `P/M` permet de calculer le produit de la matrice `P` par l'inverse de `M`.

9 Résolution des équations différentielles avec la primitive ode

Étant donnée une équation différentielle de la forme $y' = f(y,t)$ associée à une condition initiale $y(t_0) = y_0$, la primitive `ode` (pour *ordinary differential equation*) permet de calculer une valeur approchée de l'unique solution du système satisfaisant la condition initiale. Cette primitive utilise une méthode numérique d'approximation de la solution dont le principe général est analogue à celui de la méthode d'Euler. Plus précisément, il faudra spécifier lors de l'appel de la primitive `ode` quel est l'intervalle de temps sur lequel on souhaite résoudre l'équation, et quel est le pas souhaité.

On utilisera ainsi la syntaxe `y=ode(y0,t0,listet,f)` où `y0` est la position initiale, `t0` l'instant initial, `listet` l'ensemble des instants t_i auxquels on cherche une valeur approchée de la solution (en pratique `listet` sera de la forme `[t0:pas:tmax]`), `f` est la fonction définissant l'équation différentielle. La variable `y` contiendra alors une liste dont les termes sont des valeurs approchées des $\tilde{y}(t_i)$, où $\tilde{y}$ est la solution théorique de l'équation.