L'objectif de ce TP est d'acquérir les bases d'un logiciel de calcul scientifique, plus précisément du langage de programmation Scilab. Ce logiciel est utilisé pour la résolution numérique de nombreux problèmes mathématiques comme la résolution de systèmes linéaires, la résolution d'équations différentielles ou aux dérivées partielles, le calcul d'éléments propres d'une matrice, et bien d'autres.

Scilab est téléchargeable gratuitement à l'adresse suivante : `http://www.scilab.org/fr`.

Scilab est un langage interprété et ne nécessite donc pas d'étape de compilation, contrairement à d'autres langages comme le C. Un avantage d'un langage interprété est sans nul doute la rapidité du développement, des tests pouvant être réalisés facilement. De nombreuses fonctions sont préprogrammées, et des outils de représentation graphique sont disponibles.

1 Premiers pas

1.1 Une session Scilab

Il est vivement recommandé de créer un répertoire de travail pour chaque nouveau projet, et donc pour chaque séance de TP. Les différents fichiers créés seront ainsi enregistrés dans ce répertoire.

1.2 Calculatrice

L'exercice suivant permet de découvrir la première fonctionnalité de Scilab, à savoir celle d'une calculatrice scientifique.

Exercice 1
Lancer Scilab et taper les instructions suivantes dans la fenêtre de commande.

```
a = 2 + 2
b = 6 * 7;
(1 + %i)^2
```

Si la ligne d'instructions se termine par un point-virgule, alors le résultat n'est pas affiché. Plusieurs instructions peuvent figurer sur une même ligne si elles sont séparées par une virgule ou un point-virgule. Tout ce qui suit le symbole `//` définit un commentaire, ce qui permet de décrire ce que fait un morceau de code, ou bien de rendre inactive une ligne de code.

1.3 Aide en ligne

Scilab comporte un très grand nombre d'opérateurs, de commandes et de fonctions. L'utilisateur peut se référer à une aide en ligne efficace et détaillée pour mieux s'y retrouver. La commande **help** permet d'ouvrir cette aide. Suivie d'un mot-clé, cette même commande permet d'obtenir de l'aide sur la fonction recherchée (par exemple : **help sqrt**). La commande **apropos** mot−cle permet d'obtenir la liste des pages d'aide contenant le mot-clé recherché.

Exercice 2
Trouver la description de la commande **spec** . Trouver avec l'aide en ligne la fonction Scilab qui permet de calculer le déterminant d'une matrice, et celle qui permet de construire une matrice diagonale.

Les commandes **clear**, clc et clf permettent d'effacer, respectivement, les données mises en mémoire, l'écran de commande et les figures.

2 Types de données et variables

2.1 Particularités

Scilab différencie les majuscules et les minuscules. Ainsi, B et b désignent deux variables distinctes. Dans Scilab, toute variable est considérée comme un tableau d'éléments d'un type donné :

— un scalaire est un tableau à une ligne et une colonne,

— un vecteur est un tableau à une ligne ou une colonne,

— une matrice est un tableau à plusieurs lignes et plusieurs colonnes.

Cela fait de Scilab un langage axé vers le calcul matriciel. Toutes les fonctions et opérations relatives aux tableaux sont optimisées, et sont donc à utiliser aussi souvent que possible.

2.2 Types de données

Les quatre types de variables utilisés sont les types réel, complexe, chaîne de caractères et logique. On ne déclare pas le type d'une variable, qui est de fait implicite et déterminé à partir des valeurs affectées à la variable. Par exemple :

```
x = 2;
z = 2 + 3*%i;
s = 'bonjour';
```

définissent respectivement une variable de type réel, complexe et chaîne de caractères. Le type logique possède deux formes : T pour « vrai » et F pour « faux ». Un résultat de type logique est souvent utilisé dans le cas de tests.

```
x = 14;
y = 62;
test = (x == y)
```

2.3 Vecteurs

2.3.1 Quelques bases

On définit un vecteur en listant ses éléments entre crochets. Un vecteur ligne peut s'obtenir en séparant ses éléments par des espaces ou des virgules :

```
u = [2 14 6];
v = [2, 14, 6];
```

tandis qu'un vecteur colonne s'obtient en séparant ses éléments par un point-virgule :

```
w = [2; 14; 6];
```

Dans les deux cas, la longueur d'un vecteur v est donnée par **length**(v). La commande v' permet d'obtenir le vecteur transposé de v. Une subtilité apparaît lors de la manipulation de

nombres complexes : dans ce cas, `v'` donne le vecteur transconjugué de `v`. Le vecteur transposé d'un vecteur complexe peut être obtenu avec l'expression `v.'`.

Les éléments d'un vecteur peuvent être manipulés grâce à leur indice. Ainsi, `X(k)` désigne le k-ième élément du vecteur `X` (le premier élément ayant pour indice `1`).

Exercice 3
Créer un vecteur ligne `u` et un vecteur colonne `v` avec au moins trois valeurs pour chaque vecteur. Créer la variable :

```
X = [u, v']
```

qui consiste à concaténer deux vecteurs. Que permettent d'obtenir les commandes suivantes ?

```
X(3)
X(4:6)
X(1:2:5)
pos = [1,3,6];
X(pos)
```

Scilab dispose de moyens très simples pour créer des listes. La commande `a:h:b` crée une liste dont les éléments sont $a, a+h, a+2h, \ldots, a+nh$ où $n \in \mathbb{N}$ tel que $|a+nh| \leq |b|$ et $|a+(n+1)h| > |b|$. Le pas h peut être omis : `a:b` est un raccourci pour `a:1:b`. On peut aussi utiliser la commande **linspace**(a, b, n) pour créer une liste de n éléments formant une suite arithmétique de premier terme a et de dernier terme b. Ceci permet d'obtenir une discrétisation de l'intervalle $[a, b]$.

Les fonctions suivantes permettent de calculer les normes usuelles d'un vecteur.

```
norm(v, p) // norme p du vecteur v
norm(v) // norme 2 du vecteur v
norm(v, 'inf') // norme infinie du vecteur v
norm(v, %inf) // norme infinie du vecteur v
```

2.3.2 Vecteurs spéciaux

Quelques commandes permettent de générer des vecteurs de façon automatique. Ainsi la commande **ones**(1, n) permet de créer un vecteur ligne de longueur n dont tous les éléments sont égaux à 1. De la même façon, la commande **zeros**(m, 1) permet de créer un vecteur colonne de taille m dont tous les éléments sont nuls. Enfin, la commande **rand**(1, n) génère un vecteur ligne de longueur n dont tous les éléments sont des nombres aléatoires entre 0 et 1.

2.3.3 Opérations sur les vecteurs

Comme dit précédemment, Scilab est axé vers le calcul matriciel et propose ainsi différentes opérations vectorielles :

- `a*x` multiplie tous les éléments du vecteur `x` par le scalaire `a`,
- `x+y` additionne deux vecteurs `x` et `y` de même longueur,
- `x.*y` multiplie deux vecteurs `x` et `y` de même longueur composante par composante.

Exercice 4
Calculer le produit scalaire des vecteurs

$$u = \begin{pmatrix} -1 \\ -2 \\ \vdots \\ -9 \end{pmatrix} \quad \text{et} \quad v = \begin{pmatrix} 2 \\ 3 \\ \vdots \\ 10 \end{pmatrix}$$

en utilisant uniquement les opérations de base.

2.4 Matrices

2.4.1 Bases

Une matrice peut être définie avec la même syntaxe que celle utilisée pour les vecteurs. Les éléments d'une même ligne sont séparés par des espaces ou des virgules, et les colonnes sont séparées par des points-virgules. Ci-dessous, un exemple de matrice de 3 lignes et de 4 colonnes :

```
M = [2  4  0  6; 0  7  13  1; 0  2  2  1]
```

Comme pour les vecteurs, on peut obtenir facilement des informations sur une matrice :

- `M(l, c)` renvoie l'élément de la ligne `l` et de la colonne `c` de la matrice `M`,
- **size**`(M, 1)` renvoie le nombre de lignes de `M`,
- **size**`(M, 2)` renvoie le nombre de colonnes de `M`,
- `M(l, :)` renvoie la `l`-ième ligne de `M`,
- `M(:, c)` renvoie la `c`-ième colonne de `M`.

2.4.2 Opérations élémentaires

Scilab propose une syntaxe simple pour les opérations élémentaires du calcul matriciel. Ainsi, sous réserve que les dimensions soient compatibles, les opérateurs `+`, `-` et `*` correspondent aux opérations usuelles sur les matrices. On peut ajouter à ces opérations `A.*B` qui permet de multiplier les matrices `A` et `B` coefficient par coefficient. Le carré d'une matrice est obtenu avec `A^2`. Cette dernière opération peut également être faite coefficient par coefficient avec `A.^2`.

2.4.3 Matrices spéciales

Quelques matrices parmi les plus utilisées :

- **eye**`(n, n)` est la matrice identité de taille $n \times n$,
- **ones**`(n, m)` est une matrice à n lignes et m colonnes ne contenant que des 1,
- **zeros**`(n, m)` est une matrice à n lignes et m colonnes ne contenant que des 0.

2.4.4 Quelques fonctions

Différentes fonctions sont applicables aux matrices :

- `M'` désigne la matrice transposée de `M`, ou l'adjointe dans le cas d'une matrice à coefficients complexes (dans ce cas, la syntaxe `M.'` permet d'obtenir la transposée),

— **det**(M) renvoie le déterminant de M,

— **inv**(M) renvoie l'inverse de la matrice M,

— **norm**(M) renvoie la norme matricielle de M.

On peut également citer d'autres fonctions dont on laissera le soin au lecteur de déterminer le résultat : **sum**, **prod**, **max**, **min**, **mean**.

Exercice 5

Définir la matrice

$$A = \begin{pmatrix} 1 & 2 & 1 & 8 \\ 2 & 5 & 6 & 12 \\ 9 & 8 & 19 & 1 \\ 6 & 5 & 9 & 2 \end{pmatrix}.$$

1. Extraire la première ligne, la troisième colonne et l'élément situé à la deuxième ligne et à la quatrième colonne de A.

2. Échanger les deuxième et troisième lignes de A.

3. Extraire la diagonale et les parties triangulaires supérieure et inférieure de A.

Exercice 6

1. En utilisant les fonctions **ones** et **diag**, définir la matrice unité de taille 10×10.

2. Définir la matrice de taille 10×10 suivante :

$$L = \begin{pmatrix} 2 & -1 & & \\ -1 & \ddots & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 2 \end{pmatrix}.$$

2.5 Résolution de systèmes linéaires

La commande Scilab \ (backslash) est la commande générique pour résoudre un système linéaire dont la matrice est carrée. Scilab utilise une factorisation LU avec pivot partiel, suivie de la résolution des deux systèmes triangulaires qui en résultent.

```
A = [6  13; 7  5];
b = [2  1]';
x = A\b              // résolution du système
A*x-b                // vérification du résultat
```

Exercice 7

Soient

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 5 & 9 & 6 \\ 2 & 3 & 8 \end{pmatrix} \quad \text{et} \quad b = \begin{pmatrix} 2 \\ 2 \\ 1 \end{pmatrix}.$$

Résoudre le système linéaire $Ax = b$.

3 Programmation

3.1 Premier exemple de script

Les commandes Scilab peuvent être tapées directement dans la fenêtre de commandes. Cependant, elles peuvent aussi être écrites dans des fichiers, appelés scripts, qui seront par la suite exécutés par Scilab. L'utilisation de scripts comportent plusieurs avantages, comme garder une trace de ce qu'on a fait, modifier une série d'instructions parfois longue et complexe ou encore utiliser des paramètres.

Nous allons écrire notre premier script. Pour cela :

1. créer un nouveau fichier qui sera enregistré sous le nom `test.sce` et qui aura pour contenu :

```
clear;
A = rand(3, 3);
b = ones(3, 1);
x = A\b;
s = norm(A*x - b);
x, s
```

2. pour l'exécuter, taper la commande `exec('test.sce');`.

3.2 Premier exemple de fonction

On peut également créer des fichiers de fonctions, qui porteront alors l'extension `*.sci`. Un tel fichier aura la forme :

```
function [s1, s2, ..., sn] = nomFonction(e1, e2, ..., em)
    instruction1;
    instruction2;
    ...
endfunction
```

où :

— `s1, s2, ..., sn` sont les variables de sortie de la fonction,

— `e1, e2, ..., em` sont les variables d'entrée de la fonction,

— les instructions constituent le corps de la fonction.

Comme premier exemple, nous allons créer une fonction `carre()`. Pour cela :

1. créer un fichier `carre.sci` avec pour contenu :

```
function c = carre(x)
    c = x.*x;
endfunction
```

2. charger la fonction en exécutant le fichier : `exec('carre.sci');`.

Notre fonction peut alors être utilisée comme n'importe quelle autre :

```
c5 = carre(5)
n = [3 5 6];
r = carre(n)
```

ce qui peut être fait dans la fenêtre de commande, dans un script, ou même dans une autre fonction.

Exercice 8

Écrire une fonction `racines()` qui calcule les racines d'un polynôme du second degré à coefficients réels.

3.3 Les instructions conditionnelles

Comme son nom l'indique, une instruction conditionnelle permet d'exécuter une instruction ou une série d'instructions sous une condition définie à l'avance.

3.3.1 La construction if then else

L'instruction conditionnelle la plus simple se présente sous la forme suivante :

```
if expression logique then
    instructions à exécuter sous cette condition
end
```

Si la condition est vérifiée, les instructions seront exécutées. Dans le cas contraire, le corps de la condition sera ignoré. Il est possible d'imposer l'exécution d'une autre série d'instructions dans le cas où la condition n'est pas vérifiée :

```
if expression logique then
    instructions à exécuter sous cette condition
else
    instructions à exécuter si la condition n'est pas vérifiée
end
```

Enfin, il est possible de proposer différentes alternatives :

```
if expression logique 1 then
    instructions 1
elseif expression logique 2 then
    instructions 2
elseif expression logique 3 then
    instructions 3
...
else
    instructions à exécuter si la condition n'est pas vérifiée
end
```

Une expression logique consiste généralement à comparer la valeur d'une variable à une autre ou à une valeur fixée. On utilisera pour cela les opérateurs de comparaison `==` (égal à), `<`, `>`, `<=`, `>=`, `~=` et `<>` (ces deux derniers étant équivalents et servant à tester la différence). Une expression logique pourra également être une variable de type booléen : si la variable vaut `T`, la condition est vérifiée, si elle vaut `F`, ce n'est pas le cas. Enfin, une expression logique pourra aussi être la valeur (`T` ou `F`) qu'une fonction retourne.
Par exemple, en imaginant que l'on dispose d'une variable `x` :

```
if x > 0 then
    y = -x;
else
```

```
    y = x;
end
```

Exercice 9
Créer une fonction permettant de générer une matrice de taille 4×4 en fonction de la valeur de son paramètre d'entrée `numex`. Si `numex` vaut 1, alors la matrice devra être l'identité. Si `numex` vaut 2, on renverra une matrice tridiagonale. Dans tous les autres cas, la matrice ne comportera que des coefficients aléatoires, compris entre 0 et 1.

3.3.2 La construction select case

Si la série de conditions ne fait que tester différentes valeurs pour une variable donnée, on peut utiliser la syntaxe suivante :

```
select variable_a_tester
    case valeur1 then
        instructions 1
    case valeur2 then
        instructions 2
    ...
    case valeurN then
        instructions N
    else
        instructions par défaut
end
```

Scilab va alors tester, les unes après les autres, les valeurs de la variable `variable_a_tester`. Dès qu'une valeur est trouvée, la séquence d'instructions correspondante est exécutée et le reste est ignoré. La partie **else**, facultative, est là pour exécuter une série d'instructions si aucune valeur n'a été prévue.
Exercice 10
Réécrire la fonction de l'exercice 9 en utilisant la construction **select** ... **case**.

3.4 Les boucles

Une boucle est une série d'instructions qui sera exécutée un certain nombre de fois, sous différentes conditions.

3.4.1 La boucle while

Une possibilité pour réaliser une boucle est de demander l'exécution d'une série d'instructions tant qu'une condition est vérifiée, grâce au mot-clé **while** :

```
while condition
    instructions
end
```

Par exemple :

```
x = 1.1;
while x < 10
    x = x * 2;
```

```
end
```

Afin d'éviter que la boucle ne tourne indéfiniment, il est primordial que le corps de la boucle agisse sur le résultat de l'expression logique testée. L'exemple suivant montre qu'une faute de frappe peut parfois nous faire attendre longtemps.

```
n = 1;
while n < 10
    m = n + 1;
end
```

Exercice 11
Écrire une fonction permettant de calculer la factorielle d'un entier à l'aide d'une boucle **while**.

3.4.2 La boucle for

Une boucle **for** permet d'exécuter une série d'instructions un certain nombre de fois. La syntaxe est la suivante :

```
for indice = borneinf:bornesup
    instructions
end
```

où :

- `indice` est une variable qui pourra être utilisée dans la boucle,
- `borneinf` et `bornesup` sont deux constantes réelles : les paramètres de la boucle.

Durant le premier tour de la boucle, la variable `indice` vaut `borneinf`. La valeur de `indice` sera ensuite incrémentée de 1, et ce à chaque tour de boucle, jusqu'à ce que la valeur `bornesup` soit atteinte. Il est possible de modifier le pas d'incrément avec la syntaxe `borneinf:pas:bornesup`, et ce pas peut être négatif. De façon plus générale, la boucle **for** permet de parcourir toutes les valeurs d'un vecteur ligne, une à une.
Un exemple de l'utilisation de la boucle **for** est disponible ci-dessous, et permet de générer une matrice.

```
n = 3;
A = zeros(n, n);
for l = 1:n
    for c = 1:n
        A(l, c) = 1 / (l+c-1);
    end
end
```

Exercice 12
Écrire une fonction permettant de calculer la factorielle d'un entier à l'aide d'une boucle **for**.

4 Représentation graphique

Pour tracer la courbe d'une fonction `f` sur un intervalle $[a, b]$, on utilise la fonction **plot**(), comme dans l'exemple ci-dessous.

```
N = 50;                          // nombre de points de discrétisation
dx = (b - a) / (N - 1);
x = [a:dx:b];
plot(x, f(x));
```

La fonction **plot**() permet de tracer un ensemble de points de coordonnées (x_i, y_i), pour $1 \leq i \leq N$.

On peut spécifier à Scilab la couleur d'une courbe, le style du trait, ou encore le symbole représentant chaque point. Il est possible de tracer plusieurs courbes sur la même figure, et d'ajouter une légende pour y voir plus clair. Par exemple :

```
clf();
x = linspace(-1, 1, 50)';
y = x.^2;
z = 1 - x.^2;
w = 2*y;
plot(x, y, '--b', x, z, '-r', x, w, '-og');
title('Tracé de courbes avec Scilab');
xlabel('x');
ylabel('y');
legend('x^2', '1 - x^2', '2x^2');
```

Exercice 13
Représenter sur un graphique la fonction $x \mapsto \sin(2\pi x)$ calculée en 100 points de l'intervalle $[0, 1]$. Ajouter au graphique une légende. Changer la couleur du dessin ainsi que la forme du tracé.

Il est également possible de dessiner des graphes de fonctions de $\mathbb{R}^2$ dans $\mathbb{R}$. Pour visualiser la surface générée par de telles fonctions, on peut utiliser la fonction surf(), qui nécessite d'avoir auparavant généré une grille.

```
x = linspace(0, 1, 50);
y = linspace(0, 1, 50);
[X, Y] = meshgrid(x, y);
Z = sin(2 * %pi * X) .* cos(2 * %pi * Y);
surf(X, Y, Z);
```

On peut de la même façon dessiner les lignes de niveaux de la fonction avec **contour**().

```
nz = 10;                         // nombre de niveaux
contour(x, y, Z, nz);
```