ISIMA Deuxième Année F4

COURS DE MATLAB

Jonas KOKO

©2020 Institut Supérieur d'Informatique, de Modélisation et de leurs Applications Campus des Cézeaux
B.P. 1025 - 63173 AUBIERE CEDEX
http://www.isima.fr

Table des matières

1 Prise en main **13**
1.1 Démarrage et aide 13
1.2 Calculatrice 16
1.3 Ponctuation, commentaires, interruption 17
1.4 Variables 18
1.5 Gestion de la mémoire 19
1.6 Répertoire de travail 20
1.7 Sauvegarde de l'environnement de travail 20

2 Vecteurs et Matrices **21**
2.1 Création d'un vecteur ou d'une matrice 21
2.1.1 Vecteurs 21
2.1.2 Matrices 23
2.1.3 Quelques matrices prédéfinies 25
2.2 Opérations sur les matrices 27
2.2.1 Opérations globales 27
2.2.2 Opérations élément par élément 31
2.3 Manipuler les matrices 32
2.3.1 Conversion matrice/vecteur 32
2.3.2 Extraction, extension 33
2.3.3 Suppression d'une ligne ou d'une colonne 36
2.3.4 La fonction `diag` 36
2.3.5 Matrices par blocs 38
2.4 Normes vectorielles et matricielles 38
2.5 Fonction mathématiques élémentaires 40
2.6 Exercices 40

3 Algèbre linéaire **43**
3.1 Systèmes linéaires 43
3.1.1 Systèmes carrés 43
3.1.2 Systèmes rectangulaires 45
3.2 Inverses et déterminants 47
3.3 Valeurs propres, vecteurs propres 49
3.4 Factorisations de matrices 50

3.4.1 Factorisation LU . . . 51
3.4.2 Factorisation de Cholesky . . . 53
3.4.3 Factorisation QR . . . 54
3.4.4 Décomposition en valeurs singulières . . . 55
3.5 Exercices . . . 56

4 Graphisme 59
4.1 Généralités . . . 59
4.2 Graphisme 2D . . . 59
4.2.1 Les axes . . . 61
4.2.2 Les commentaires . . . 63
4.2.3 Décomposition de la fenêtre en sous-fenêtres . . . 64
4.2.4 Sauvegarder une figure . . . 65
4.3 Visualisation 3D . . . 66
4.3.1 Courbes 3D . . . 66
4.3.2 Surface analytique . . . 67
4.3.3 Surface définie par un ensemble de points . . . 69
4.3.4 Courbes de niveau . . . 71
4.4 Exercices . . . 75

5 Matrices creuses et méthodes itératives 77
5.1 Mode de stockage . . . 77
5.2 Création . . . 77
5.3 Opérations sur les matrices creuses . . . 80
5.4 Factorisation et méthodes directes . . . 80
5.4.1 Factorisation LU . . . 80
5.4.2 Factorisation de Cholesky . . . 82
5.5 Permutation des lignes et des colonnes . . . 82
5.6 Factorisation incomplète et préconditionnement . . . 85
5.6.1 Factorisation LU incomplète . . . 86
5.6.2 Factorisation de Cholesky incomplète . . . 87
5.7 Méthodes itératives . . . 88
5.7.1 Méthode du gradient conjugué préconditionné . . . 88
5.7.2 Méthode GMRES . . . 91
5.8 Exercices . . . 93

6 Programmation avec MATLAB 95
6.1 Types de données et variables . . . 95
6.2 Structures de données avancées . . . 96
6.2.1 Tableaux de cellules (cell arrays) . . . 96
6.2.2 Structures . . . 97
6.3 Scripts . . . 97
6.4 Fonctions . . . 97
6.5 Structures de contrôle . . . 103
6.5.1 La boucle `for` . . . 103
6.5.2 La boucle `while` . . . 103
6.5.3 L'alternative `if` . . . 104
6.5.4 Les expressions logiques . . . 105
6.5.5 Instructions de rupture de séquence . . . 106

6.6 Quelques fonctions internes . . . 107
6.6.1 Les fonctions logiques . . . 107
6.6.2 La fonction `find` . . . 108
6.6.3 Fonctions de réduction . . . 109
6.6.4 Opérations sur les ensembles . . . 113
6.7 Importation/exportation de données . . . 116
6.7.1 Les fonctions `save` et `load` . . . 116
6.7.2 Les fonctions `fprintf` et `fscanf` . . . 117
6.8 Optimisation d'un code . . . 118
6.9 Exercices . . . 121

Table des figures

3.1 Régression linéaire 47

4.1 Graphe de la fonction $y = x \sin x \cos x$ 60
4.2 Graphes de $y = x \sin x \cos x$, $y = \mathrm{e}^{-x^2+\sin x}$ et $y = \mathrm{e}^{\sin x}$ 62
4.3 Axes légendés avec des formules mathématiques. 62
4.4 Polynômes de Tchebychev pour $n = 0$, 1, 2, 3. 63
4.5 Polynômes de Tchebychev pour $n = 1, 2, 3, 4$. 65
4.6 Courbe 3D 66
4.7 Commande `plot3(x,y,z)` avec `x`, `y` et `z` des matrices 67
4.8 Surface tracée avec `plot3` 67
4.9 Tracé fil de fer de la surface définie par (4.1) 68
4.10 Tracé fil de fer de la surface définie par (4.1), avec `view(30,60)` 69
4.11 Tracé fil de fer d'une surface obtenue par un ensemble de points 70
4.12 Nuage de points 70
4.13 Triangulation de Delaunay d'un ensemble de points 71
4.14 Tracé par facettes d'une surface définie sur la triangulation 4.13 72
4.15 Tracé par facettes d'une surface définie sur la triangulation 4.13 72
4.16 Courbes de niveau 73
4.17 Courbes de niveau avec valeurs 74
4.18 Courbes de niveau avec valeurs sélectionnées 74
4.19 Trajectoires z_1, z_2 et z_3 dans le plan 75
4.20 Lignes de niveaux sur une surface 76

5.1 Visualisation de la matrice creuse `west0479` 79
5.2 Visualisation des matrices L et U de la factorisation LU de `west0479` 81
5.3 Matrice `west0479` permutée avec `symrcm` 83
5.4 Matrice du Laplacien 85
5.5 Matrice du Laplacien permutée avec `symamd` 86
5.6 Historique du résidu relatif pour `pcg` 91

Liste des tableaux

1.1 Listes de mots réservés 19
1.2 Constantes et variables prédéfinies 19

2.1 Quelques matrices prédéfinies disponibles 25
2.2 Opérateurs arithmétiques élémentaires 28
2.3 Fonctions mathématiques pour les complexes 40
2.4 Quelques fonctions trigonométriques 41
2.5 Exponentielles et logarithmes 41

3.1 Tableau des observation du modèle linéaire 45
3.2 Quelques fonctions de factorisation de matrices 50
3.3 Tableau des observations du modèle quadratique de l'exercice 3.2 57

4.1 Couleur, symbole et style d'un tracé 61

5.1 Solveurs itératifs 89

6.1 Opérateurs de comparaison de MATLAB 105
6.2 Opérateurs logiques de MATLAB 106
6.3 Instructions de rupture de séquence 106
6.4 Fonctions logiques 107

Liste des programmes

6.1 Fonction matrice de distance avec des boucles `for` 119
6.2 Fonction matrice de distances vectorisée 120

CHAPITRE 1

PRISE EN MAIN

1.1 Démarrage et aide

Sur Unix et dérivées, le démarrage de Matlab se fait simplement en tapant `matlab` dans une fenêtre de commande. Sur Windows, il suffit de cliquer sur l'icône Matlab. Dans les deux cas, on obtient sur la fenêtre de commande le prompt >>, caractéristique de Matlab.

L'aide en ligne s'obtient en tapant la commande `help`. Une longue liste de sujets, pour lesquels l'aide est disponible, apparaît alors.

```
>> help

HELP topics:

matlab/general       -  General purpose commands.
matlab/ops           -  Operators and special characters.
matlab/lang          -  Programming language constructs.
matlab/elmat         -  Elementary matrices, matrix manipulation.
matlab/elfun         -  Elementary math functions.
matlab/specfun       -  Specialized math functions.
matlab/matfun        -  Matrix functions, numerical linear algebra.
matlab/datafun       -  Data analysis and Fourier transforms.
matlab/audio         -  Audio support.
matlab/polyfun       -  Interpolation and polynomials.
matlab/funfun        -  Function functions and ODE solvers.
matlab/sparfun       -  Sparse matrices.
matlab/graph2d       -  Two dimensional graphs.
matlab/graph3d       -  Three dimensional graphs.
matlab/specgraph     -  Specialized graphs.
matlab/graphics      -  Handle Graphics.
matlab/uitools       -  Graphical user interface tools.
matlab/strfun        -  Character strings.
matlab/iofun         -  File input/output.
matlab/timefun       -  Time and dates.
matlab/datatypes     -  Data types and structures.
matlab/verctrl       -  Version control.
```

```
matlab/demos         -  Examples and demonstrations.
toolbox/local        -  Preferences.
toolbox/optim        -  Optimization Toolbox.
toolbox/pde          -  Partial Differential Equation Toolbox.

For more help on directory/topic, type "help topic".
```

Après, il suffit d'exécuter

>>help *sujet*

pour avoir l'aide sur le sujet concerné. Pour avoir la description d'une fonction MATLAB, il suffit d'exécuter la commande `help` avec le nom de la fonction

>>help *fonction*

Par exemple, si on veut la description de la fonction `plot`, qui permet de tracer des courbes 2D, on tape simplement

```
>>help plot

 PLOT   Linear plot.
    PLOT(X,Y) plots vector Y versus vector X. If X or Y is a matrix,
    then the vector is plotted versus the rows or columns of the matrix,
    whichever line up.  If X is a scalar and Y is a vector, length(Y)
    disconnected points are plotted.

    PLOT(Y) plots the columns of Y versus their index.
    If Y is complex, PLOT(Y) is equivalent to PLOT(real(Y),imag(Y)).
    In all other uses of PLOT, the imaginary part is ignored.

    Various line types, plot symbols and colors may be obtained with
    PLOT(X,Y,S) where S is a character string made from one element
    from any or all the following 3 columns:

           y     yellow        .     point              -     solid
           m     magenta       o     circle             :     dotted
           c     cyan          x     x-mark             -.    dashdot
           r     red           +     plus               --    dashed
           g     green         *     star
           b     blue          s     square
           w     white         d     diamond
           k     black         v     triangle (down)
                               ^     triangle (up)
                               <     triangle (left)
                               >     triangle (right)
                               p     pentagram
                               h     hexagram

    For example, PLOT(X,Y,'c+:') plots a cyan dotted line with a plus
```

```
    at each data point; PLOT(X,Y,'bd') plots blue diamond at each data
    point but does not draw any line.

    PLOT(X1,Y1,S1,X2,Y2,S2,X3,Y3,S3,...) combines the plots defined by
    the (X,Y,S) triples, where the X's and Y's are vectors or matrices
    and the S's are strings.

    For example, PLOT(X,Y,'y-',X,Y,'go') plots the data twice, with a
    solid yellow line interpolating green circles at the data points.

    The PLOT command, if no color is specified, makes automatic use of
    the colors specified by the axes ColorOrder property.  The default
    ColorOrder is listed in the table above for color systems where the
    default is yellow for one line, and for multiple lines, to cycle
    through the first six colors in the table.  For monochrome systems,
    PLOT cycles over the axes LineStyleOrder property.

    PLOT returns a column vector of handles to LINE objects, one
    handle per line.

    The X,Y pairs, or X,Y,S triples, can be followed by
    parameter/value pairs to specify additional properties
    of the lines.

    See also SEMILOGX, SEMILOGY, LOGLOG, GRID, CLF, CLC, TITLE, XLABEL,
    YLABEL, AXIS, AXES, HOLD, COLORDEF, LEGEND, SUBPLOT, and STEM.
```

Il est possible de retrouver toutes les fonctions contenant un mot clé. Pour cela on utilise la commande `lookfor`. Par exemple, si on recherche les fonctions relatives à la factorisation de matrices

```
>> lookfor factorization
CHOL   Cholesky factorization.
CHOLUPDATE Rank 1 update to Cholesky factorization.
LU     LU factorization.
QRDELETE Delete column from QR factorization.
QRINSERT Insert column in QR factorization.
QRUPDATE Rank 1 update to QR factorization.
QZ     QZ factorization for generalized eigenvalues.
CHOLINC  Sparse Incomplete Cholesky.
LUINC  Sparse Incomplete LU factorization.
SYMBFACT Symbolic factorization analysis.
```

Remarque 1.1 *Si dans l'aide en ligne, les fonctions et commandes* MATLAB *apparaissent en lettres majuscules c'est seulement pour les distinguer des autres mots. Les fonctions et commandes Matlab doivent être codées en minuscules.*

Les commandes d'éditions de Matlab sont simples:

↑ remonter dans l'historique de commandes;

↓ descendre dans l'historique de commandes;

←, → déplacement sur la ligne;

Backspace, Delete modifications sur la ligne de commande.

1.2 Calculatrice

Une ligne de commande de Matlab est de la forme [1]

```
[variable=]expression[;]
```

Lorsque le point-virgule (`;`) est absent en fin de ligne, la valeur de l'expression est affichée à l'écran. Les sorties Matlab sont donc en grande partie gérées par le point-virgule. Lorsqu'il n'y a pas de variable pour récupérer la valeur de l'expression, c'est la variable par défaut de MATLAB, `ans`, qui est utilisée. Voici quelques exemples de ligne de commande

```
>> 2*pi/9
ans =
    0.6981

>> x=6^10
x =
    60466176
```

Par défaut les réels sont affichés [2] en simple précision (5 chiffres significatifs). On peut modifier l'affichage avec la commande `format`.

```
>> pi
ans =
    3.1416

>> format long
>> pi
ans =
   3.14159265358979

>> format short
>> pi
ans =
    3.1416
```

Pour obtenir un affichage de réels en virgule flotante, il suffit de rajouter `e` dans à la commande `format`.

```
>> format short e
>> pi^6
ans =
   9.6139e+02
```

1. Tout ce qui apparaît entre `[]` est, ici, optionnel.
2. Tous les calculs dans Matlab sont en double précision

Enfin, `format short g` ou `format long g` laisse le choix à MATLAB d'utiliser le «meilleur» affichage (virgule fixe ou virgule flottante) en fonction du réel.

Les nombres complexes se codent de façon naturelle avec `i` ou `j` comme imaginaire pure.

```
>> 1.5+i
ans =
   1.5000 + 1.0000i

>> c=sqrt(-1)
c =
   0 + 1.0000i
```

Les opérateurs sont les mêmes que pour les nombres réels.

```
>> c1=exp(i)
c1 =
   0.5403 + 0.8415i

>> c2=sqrt(-2)
c2 =
    0 + 1.4142i

>> c1+c2
ans =
   0.5403 + 2.2557i

>> c1*c2
ans =
    -1.1900 + 0.7641i

>> c1/c2
ans =
    0.5950 - 0.3821i
```

1.3 Ponctuation, commentaires, interruption

Une ligne de commande MATLAB peut comporter plusieurs commandes séparées par des virgules (`,`) ou des points-virgules (`;`).

```
>> x=3/8, y=2.2678/37.80; z=(x+y)/sqrt(-4)
x =
   0.3750

z =
   0 - 0.2175i
```

Une ligne de commande MATLAB peut comporter des commentaires signalés par le symbole `%`. Tout ce qui suit ce symbole est alors ignoré par l'interpréteur MATLAB.

```
>> x=pi         % nombre réel
x =
    3.1416

>> z=sqrt(-1)   % nombre complexe
z =
        0 + 1.0000i
```

Si une expression est trop longue pour tenir sur une ligne de commande, alors on termine la ligne par `...` (3 points).

```
>> y=cos(2*pi/3)*(sin(5.78*pi/4)^2+...
     cos(pi/5)^2)

y =
   -0.8125
```

Un traitement MATLAB peut être à tout moment interrompu en appuyant simultanément sur les touches **Ctrl** et **C**.

1.4 Variables

Les noms des variables MATLAB sont des mots de 63 caractères maximum, sans espace et sans symbole de ponctuation. Le nom d'une variable doit obligatoirement commencer par une lettre de l'alphabet. Il y a une distinction entre majuscules et minuscules pour les noms de variables.

```
>> x1=3.8675, X1=pi^2
x1 =
    3.8675

X1 =
    9.8696
```

Comme tous les langages, MATLAB dispose de mots-clés. Certains mots-clés ne peuvent être utilisés comme variables. Ce sont les mots réservés du langage. La liste des mots-clés de MATLAB est donnée dans le tableau 1.1.

```
>> for=2.16
??? for=2.16
       |
Error: "identifier" expected, "=" found.
```

La liste de de ces mots réservés est fournie par la fonction `iskeyword`. D'autres mots-clés, non réservés, représentent des variables ou des constantes prédéfinies, *cf.* tableau 1.2. Ces mots-clés peuvent être utilisés comme variables définies par l'utilisateur.

```
>> pi, realmax
ans =
     3.1416
```

```
ans =
     1.7977e+308

>> pi=278, realmax=.2568
pi =
    278

realmax =
         0.2568
```

break	case	catch	continue	else	elseif	end	for	function
global	if	otherwise	persistent	return	switch	try	while	

TABLE 1.1 – Listes de mots réservés

`ans`	*answer* variable MATLAB par défaut
`eps`	précision numérique relative, `eps=2.220446049250313e-016`
`inf`	l'infini mathématique ∞, *e.g.* `1.0/0.`
`nan`	littéralement *Not a Number*, *e.g.* `0/0`
`pi`	constante $\pi = 3.14159265358979$
`i` ou `j`	nombre complexe $\sqrt{-1}$
`nargin`	nombre d'arguments d'entrée d'une fonction
`nargout`	nombre d'arguments de sortie d'une fonction
`realmin`	plus petit réel utilisable
`realmax`	plus grand réel utilisable
`bitmax`	plus grand entier utilisable, stocké en double précision
`varargin`	nombre variable d'arguments d'entrée d'une fonction
`varargout`	nombre variable d'arguments de sortie d'une fonction

TABLE 1.2 – Constantes et variables prédéfinies

1.5 Gestion de la mémoire

La commande `who` donne la liste des variables actives dans l'espace de travail. Mais elle est moins précise. Pour avoir plus d'informations, il est préférable d'utiliser la commande `whos` qui donne non seulement les variables mais aussi leur type, leur taille (comme tableau) et l'espace mémoire occupé. Voici un exemple.

```
>> whos
  Name       Size          Bytes  Class

  A         50x50          20000  double array
  x          1x10001       80008  double array
  y          1x10001       80008  double array
  zz       100x100         80000  double array
```

Il est possible de supprimer une ou plusieurs variables avec la commande `clear`. Par exemple

```
>> clear x zz
```

supprime les variables `x` et `zz` de l'espace de travail. En particulier `clear all` efface toutes les variables de l'espace de travail.

1.6 Répertoire de travail

Avant de commencer à travailler, il faut indiquer à Matlab le répertoire dans lequel on veut travailler. Sous Unix, le répertoire de travail est le répertoire courant. Sous Windows, le répertoire par défaut s'appelle `works` et se trouve dans le répertoire d'installation de Matlab. Pour le changer il suffit d'entrer le nom de votre répertoire de travail dans la fenêtre `Current Directory` de la barre de menu.

1.7 Sauvegarde de l'environnement de travail

La fonction `save` permet de sauvegarder (par défaut en binaire) tout (par défaut) ou partie de l'espace de travail, qui se récupère grâce à la commande `load`. Par exemple si on veut sauvegarder les variables `X` et `Y` dans le fichier `FichXY` il suffit d'exécuter la ligne de commande

```
>> save FichXY X Y
```

Par défaut le fichier de sauvegarde est en binaire et comporte l'extension `.mat`.

La fonction `load` est l'inverse de la fonction `save`. Par exemple, pour récupérer les variables `X` et `Y` sauvegardées dans le fichier `FichXY`, il suffit de faire

```
>> load FichXY X Y
```

La commande `diary` permet de sauvegarder l'intégralité d'une session de travail. Elle est très utile lorsqu'on tâtonne. On démarre la sauvegarde avec `diary on` et on arrête avec `diary off`. Dans tous les cas lorsqu'on tape `diary`, on passe d'un état à un autre, *i.e.* on passe de `diary on` à `diary off` ou *vice versa*. Il est possible d'utiliser la forme fonctionnelle suivante

```
>> diary('nomfich')
```

Alors toute la session sera sauvegardée dans le fichier `nomfich`.

CHAPITRE 2

VECTEURS ET MATRICES

Comme son nom l'indique, MATLAB ne travaille qu'avec des matrices, au sens large. Ainsi, un scalaire est pour MATLAB une matrice 1×1 et un vecteur de taille n, une matrice $n \times 1$ ou $1 \times n$ suivant que le vecteur est en colonne ou en ligne. En plus des opérations usuelles sur les matrices (somme, soustraction, produit) MATLAB dispose d'opérateurs qui agissent élément par élément, permettant de se passer des boucles.

2.1 Création d'un vecteur ou d'une matrice

2.1.1 Vecteurs

Pour définir un vecteur (ligne ou colonne) il suffit de donner la liste de ses éléments entre crochets (`[]`). Pour un vecteur ligne, les éléments sont séparés au choix par un espace ou une virgule (`,`)

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

>> v=[3,6,8,1.765]
v =
    3.0000    6.0000    8.0000    1.7650
```

Pour un vecteur colonne, les éléments sont séparés au choix par un point-virgule (`;`) ou un retour-chariot.

```
>> w=[1; 2; 3]
w =
     1
     2
     3

>> w=[3.56
      2.9
      6.3456
      4.5]
w =
```

```
   3.56000
   2.90000
   6.34560
   4.50000
```

On peut utiliser l'opérateur : (énumération) pour créer un vecteur. La syntaxe générale est

```
v=debut[:pas]:fin
```

où `debut` et `fin` sont des nombres, `pas` est le pas de la tabulation. Par défaut `pas=1`.

```
>> v=-1:.5:1
v =
   -1.0000  -0.5000 0 0.5000 1.0000
```

Remarque 2.1 *Un vecteur généré par énumération est toujours un vecteur ligne.*

Il est possible de mixer les différentes formes de saisie de vecteurs.

```
>> v=[-5 3 7 0:2:8]
v =
   -5     3     7     0     2     4     6     8
```

Pour accéder à un élément d'un vecteur, on utilise son indice entre parenthèses.

```
>> v=0:10
v =
   0     1     2     3     4     5     6     7     8     9    10

>> v(7)
ans =
     6
```

On peut générer des vecteurs dont les composantes sont uniformément réparties grâce à la fonction `linspace` dont la syntaxe est

```
linspace(x1,x2,n)
```

génère un vecteur ligne de `n` composantes, subdivision uniforme de l'intervalle `[x1, x2]`.

```
>> x=linspace(0,1,5)
x =
   0    0.2500    0.5000    0.7500    1.0000
```

Par défaut, *i.e.* en l'absence du troisième argument, le vecteur généré est de taille 100. Pour avoir un vecteur ligne avec des composantes réparties suivant une échelle logarithmique, on utilise `logspace`. la syntaxe est la même que pour `linspace` sauf que par défaut, le vecteur généré est de taille 50.

```
>> x=logspace(0,1,5)
x =
   1.0000    1.7783    3.1623    5.6234   10.0000
```

Pour avoir le vecteur transposé, l'apostrophe «'» (prime) suffit.

```
>> v=[3.56; 2.9; 6.3456; 4.5]
v =
    3.5600
    2.9000
    6.3456
    4.5000

>> vt=v'
vt =
    3.5600    2.9000    6.3456    4.5000
```

La taille d'un vecteur est obtenue grâce à la fonction `length`.

```
>> v=0:pi/5:1.5*pi;
>> n=length(v)
n =
     8

>> v=[];  % vecteur vide
>> length(v)
ans =
     0
```

Pour savoir si on a affaire à un vecteur ligne ou à un vecteur colonne, on utilise la fonction `size` qui retourne le nombre de lignes et de colonnes d'un tableau passé en argument. La fonction `size` est détaillée dans la section 2.1.2.

2.1.2 Matrices

Pour définir une matrice, il suffit de séparer les lignes de la matrice par un point-virgule (`;`). Par exemple

```
>> A=[1 2 3; 4 1 2; 0 2 4]
A =
     1     2     3
     4     1     2
     0     2     4
```

On peut aussi créer une matrice directement à partir de vecteurs. Si `v1, v2, v3` sont des vecteurs lignes de *même taille*, alors `[v1; v2; v3]` est une matrice dont les lignes sont les vecteurs `v1`, `v2` et `v3`; et `[v1' v2' v3']` est une matrice dont les colonnes sont les vecteurs `v1'`, `v2'` et `v3'`.

```
>> v1=[1 2 3];
>> v2=[1 1 1];
>> v3=[3.5 7.8 0.5];
>> A=[v1; v2; v3], B=[v1' v2' v3']
A =
    1.0000    2.0000    3.0000
    1.0000    1.0000    1.0000
    3.5000    7.8000    0.5000
B =
    1.0000    1.0000    3.5000
    2.0000    1.0000    7.8000
    3.0000    1.0000    0.5000
```

Donc `B` est la transposée de `A`. Pour résumer, les lignes d'une matrice sont séparées par un point-virgule (ou un retour-chariot) et les colonnes par un espace (ou une virgule).

Comme pour les vecteurs, la matrice transposée s'obtient grâce à l'apostrophe (`'`).

```
>> A=[8 2 7; 0 1 3]
A =
   8     2     7
   0     1     3

>> B=A'
B =
   8     0
   2     1
   7     3
```

Pour accéder à un élément d'une matrice, il suffit de préciser son numéro de ligne et son numéro de colonne entre parenthèses.

```
>> A=pascal(3)
A =
   1     1     1
   1     2     3
   1     3     6

>> A(3,2)
ans =
     3
```

La fonction `size` retourne la taille d'une matrice au sens MATLAB, c'est-à-dire en nombre de lignes et de colonnes. La syntaxe est

```
[m,n]=size(A)
```

qui donne la taille $m \times n$ de la matrice A.

```
>> A=[0 2 5 6; 3 6 5 2; 1 1 2 2];
>> [m,n]=size(A)
m =
   3
n =
   4
```

A noter que si `v` est un vecteur la fonction `length(v)` est équivalente à `max(size(v))`.

Remarque 2.2 *Si* `v` *est un vecteur,* `length(v)` *retourne 0 tandis que* `size(v)` *retourne le vecteur vide,* `i.e. []`.

2.1.3 Quelques matrices prédéfinies

Fonction	Matrice générée
`zeros`	matrice nulle
`eye`	matrice identité
`ones`	matrice ne contenant que des 1
`hilb`	matrice de Hilbert
`pascal`	matrice de Pascal
`rand`	matrice aléatoire (loi uniforme)
`randn`	matrice aléatoire (loi normale)

TABLE 2.1 – Quelques matrices prédéfinies disponibles

MATLAB dispose de plusieurs matrices prédéfinies dont quelques unes sont données figure 2.1.

Matrice identité

La matrice identité d'ordre n s'obtient par la fonction `eye`.

```
>> I=eye(5)
 I =
     1     0     0     0     0
     0     1     0     0     0
     0     0     1     0     0
     0     0     0     1     0
     0     0     0     0     1
```

Matrice nulle

La fonction `zeros` permet d'avoir une matrice nulle de taille $m \times n$.

```
>> O=zeros(3,2)
O =
     0     0
     0     0
     0     0
```

Pour une matrice nulle carrée un seul argument suffit.

```
>> O=zeros(2)
O =
     0     0
     0     0
```

On peut aussi générer un vecteur (ligne ou colonne) ne contenant que des zéros

```
>> U=zeros(1,4)
U =
     0     0     0     0
```

Matrice ne contenant que des 1

La fonction `ones` retourne une matrice $m \times n$ ne contenant que des 1.

```
>> Un=ones(3,5)
Un =
     1     1     1     1     1
     1     1     1     1     1
     1     1     1     1     1
```

Comme pour la matrice nulle, un seul argument suffit pour une matrice carrée.

Matrice de Hilbert

C'est la matrice carrée $H = (h_{ij})$ d'ordre n définie par

$$h_{ij} = \frac{1}{i+j-1}, \quad i, j = 1, \ldots, n.$$

Elle est fournie par la fonction `hilb`.

```
>> H=hilb(4)
H =
    1.0000    0.5000    0.3333    0.2500
    0.5000    0.3333    0.2500    0.2000
    0.3333    0.2500    0.2000    0.1667
    0.2500    0.2000    0.1667    0.1429
```

Matrice de Pascal

C'est la matrice carrée d'ordre n, symétrique définie positive obtenue à partir du triangle de Pascal. La matrice de Pascal est entière et son inverse aussi.

```
>> P=pascal(3)
P =
   1     1     1
   1     2     3
   1     3     6

>> inv(P)          % inverse de P
ans =
     3    -3     1
    -3     5    -2
     1    -2     1
```

Matrices aléatoires

Avec MATLAB il est possible d'obtenir une matrice $m \times n$ générée de manière aléatoire soit selon une loi uniforme (fonction `rand`) soit selon une loi normale (`randn`).

```
>> U=rand(2,3), N=randn(2,3)
U =

    0.9501    0.6068    0.8913
    0.2311    0.4860    0.7621

N =
   -0.4326    0.1253   -1.1465
   -1.6656    0.2877    1.1909
```

2.2 Opérations sur les matrices

Le tableau 2.2 présente les opérateurs arithmétiques élémentaires dans l'ordre croissant de priorité. Ces opérateurs sont valables pour les scalaires, les vecteurs et les matrices, moyennant les précautions d'usage.

Remarque 2.3 *Les opérateurs `*`, `/` et `^` peuvent être précédés d'un point (i.e. `.*`, `./` et `.^`) pour signifier que l'opération s'effectue élément par élément, cf. § 2.2.2.*

Pour avoir de l'aide en ligne sur les opérateurs

`help arith` opérateurs arithmétiques élémentaires, sans les divisions;

`help slash` opérateurs de division;

`help ops` tous les opérateurs disponibles dans MATLAB.

2.2.1 Opérations globales

Les opérations globales entre matrices permettent de reproduire les opérations usuelles du calcul matriciel.

Opérateurs	Opération
+	Addition
−	Soustraction
*	Multiplication
/	Division
\	Division inverse (pour la résolution de systèmes linéaires)
^	Puissance

TABLE 2.2 – Opérateurs arithmétiques élémentaires

Addition, soustraction

L'addition ou la soustraction de deux matrices (ou vecteurs) n'est possible que si les matrices sont de même dimension ou si l'un des deux opérandes est un scalaire. Dans ce dernier cas, l'opération est effectuée entre le scalaire et tous les éléments de la matrice.

```
>> A=pascal(3), B=ones(3)
A =
   1     1     1
   1     2     3
   1     3     6

B =
   1     1     1
   1     1     1
   1     1     1

>> X=A-B
X =
   0     0     0
   0     1     2
   0     2     5

>> Y=A+B
Y =
   2     2     2
   2     3     4
   2     4     7

>> Z=B+5
Z =
   6     6     6
   6     6     6
   6     6     6
```

Produit de vecteurs

Un vecteur ligne et un vecteur colonne de même taille, peuvent être multipliés directement par l'opérateur `*`. Le résultat du produit est soit un scalaire (produit scalaire) soit une matrice.

```
>> u=[2 5 6]; v=ones(3,1);
>> x=u*v
x =
   13

>> X=v*u
X =
   2     5     6
   2     5     6
   2     5     6
```

Donc le produit scalaire de deux vecteurs colonnes s'obtient en multipliant (à gauche) l'un des deux vecteurs par la transposée de l'autre.

```
>> u=ones(3,1); v=[3; 5; 7];
>> x=u'*v
x =
   15

>> y=v'*u
y =
   15
```

On peut aussi utiliser la fonction MATLAB `dot` qui calcule le produit scalaire de deux vecteurs de même longueur. Avec les vecteurs précédents on a

```
>> dot(u,v)
ans =
     15

>> dot(u,v')
ans =
     15
```

Produit de matrices

L'opérateur `*` permet d'effectuer le produit matriciel usuel entre deux matrices. Les dimensions des deux matrices opérandes doivent donc être compatibles, *i.e.* le nombre de colonnes de la première doit être égal au nombre de lignes de la seconde.

```
>> A=pascal(3); B=[2 3 0 1; 5 7 2 1; 8 3 4 1];
>> C=A*B
C =
   15    13     6     3
```

```
   36    26    16     6
   65    42    30    10

>> Z=B*C
??? Error using ==> *
Inner matrix dimensions must agree.
```

Dans le cas où l'un des opérandes est un scalaire et l'autre une matrice, la multiplication est toujours valide. L'opération s'effectue sur chaque élément de la matrice.

```
>> A=ones(2,4);
>> B=2*A
B =
   2     2     2     2
   2     2     2     2

>> C=A*3
C =
   3     3     3     3
   3     3     3     3
```

Division

Si A et B sont des matrices, l'opération MATLAB `B/A` est équivalente à $\texttt{BA}^{-1}$. Ce qui revient à résoudre le système linéaire $XA = B$. Ce qui est très peu courant.

L'opérateur `/` est utilisé le plus souvent avec comme deuxième opérande un scalaire, le premier pouvant être un scalaire, un vecteur ou une matrice.

```
>> A=pascal(3); u=ones(1,3);
>> B=A/4
B =
   0.2500    0.2500    0.2500
   0.2500    0.5000    0.7500
   0.2500    0.7500    1.5000

>> v=u/3
v =
   0.3333    0.3333    0.3333
```

L'opérateur \ sert à résoudre les systèmes linéaires par élimination de Gauss. Pour MATLAB, le système linéaire

$$Ax = b$$

a pour solution `x = A\b`. Voici un exemple de résolution d'un système linéaire avec MATLAB par élimination de Gauss.

```
>> A=[3 0 1;0 5 -2;1 -2 7];
>> b=[1;2;3];
>> x=A\b
x =
```

```
   0.1364
   0.6364
   0.5909
```

L'opérateur \ sera étudié en détail au § 3.1.

Puissance

Si A est une matrice carrée et a un scalaire, l'opération MATLAB `A^a` est équivalente à A^a. Si a est un entier plus grand que 1, alors la matrice A est multipliée a fois pour avoir le résultat. Pour d'autres valeurs de a, le calcul fait intervenir les valeurs propres et les vecteurs propres.

```
>> A=pascal(3);
A =
   1     1     1
   1     2     3
   1     3     6

>> A^2
ans =
     3     6    10
     6    14    25
    10    25    46

>> A^(1.25)
ans =
    1.1845    1.5082    1.8773
    1.5082    3.1541    5.1705
    1.8773    5.1705    9.9251
```

2.2.2 Opérations élément par élément

Une variante très utile des opérateurs `*`, `/` et `^` permet d'effectuer les opérations entre matrices élément par élément. Il suffit simplement de rajouter un point devant l'opérateur pour que MATLAB effectue l'opération élément par élément. Soit $A = (A_{ij})$ et $B = (B_{ij})$ des matrices de mêmes dimensions $m \times n$, on a

`C=A.*B` équivalent à $C_{ij} = A_{ij} * B_{ij}, i = 1, \ldots, m, j = 1, \ldots, n$

`C=A./B` équivalent à $C_{ij} = A_{ij}/B_{ij}, i = 1, \ldots, m, j = 1, \ldots, n$

`C=A.^B` équivalent à $C_{ij} = (A_{ij})^{B_{ij}}, i = 1, \ldots, m, j = 1, \ldots, n.$

Les boucles en i et j sont donc «déroulées» indirectement par MATLAB.

```
>> A=pascal(3), B=ones(3);
A =
   1     1     1
   1     2     3
   1     3     6
```

```
>> C1=A*B
C1 =
     3     3     3
     6     6     6
    10    10    10

>> C2=A.*B
C2 =
     1     1     1
     1     2     3
     1     3     6
```

Soit à tabuler la fonction

$$y = \frac{1}{1+x^2}, \quad x \in [-1,\ 1].$$

Les commandes en ligne MATLAB pour la tabulation sont alors.

```
>> x=-1:.25:1;
>> y=1.0./(1+x.*x)
y =
   0.5000   0.6400   0.8000   0.9412   1.0000   0.9412   0.8000   0.6400   0.5000
```

On commence donc par tabuler x par simple énumération, puis on tabule y en utilisant les opérateurs `./` et `.*` . A noter que le `1.0` permet de distinguer le point signalant un réel du point de l'opérateur `./`.

2.3 Manipuler les matrices

Par manipuler nous entendons
— conversion d'un vecteur en matrice et *vice versa*;
— extraction d'une sous-matrice, extension d'une matrice;
— extraction des diagonales ou des parties triangulaires supérieure et inférieure.

2.3.1 Conversion matrice/vecteur

Une matrice peut être facilement convertie en vecteur grâce à l'opérateur `:` (deux-points). Le vecteur résultat est composé de colonnes de la matrice source mises bout à bout.

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

>> u=A(:)
u =
   1
   2
   3
```

```
   4
   5
   6
```

Pour la conversion d'un vecteur en matrice, on utilise la fonction `reshape`. La syntaxe est

```
A=reshape(u,m,n)
```

découpe le vecteur `u` en `n` vecteurs de longueurs `m` pour former la matrice `A`. La matrice est remplie colonne par colonne. Avec le vecteur `u` de l'exemple précédent on a

```
>> A=reshape(u,2,3)
A =
   1     3     5
   2     4     6

>> B=reshape(u,3,2)
B =
   1     4
   2     5
   3     6
```

2.3.2 Extraction, extension

L'opérateur `:` s'utilise aussi pour extraire un sous-vecteur ou une sous-matrice par énumération des indices concernés. La syntaxe est

```
debut:[pas:]fin
```

où `debut`, `pas` et `fin` sont des expressions entières.

```
>> v=[0:.5:pi]
v =
   0    0.5000    1.0000    1.5000    2.0000    2.5000    3.0000

>> w=v(2:5)
w =
    0.5000    1.0000    1.5000    2.0000

>> A=[1 2 3; 4 1 2; 0 2 4]
A =
   1     2     3
   4     1     2
   0     2     4

>> v=A(:,1)      % toutes les lignes de la colonne 1
v =
   1
   4
```

```
   0

>> H=A(2:3,2:3)  % H=A(i,j), 2<=i,j<=3
H =
   1     2
   2     4

>> u=A(2,:)      % toutes les colonnes de la ligne 2
u =
   4     1     2
```

Au lieu d'énumérer les indices, on peut utiliser un vecteur d'indices pour extraire un sous-vecteur

```
>> u=[0:2:10]
u =
   0     2     4     6     8    10

>> I=[2 5 6];
>> v=u(I)
v =
   2     8    10
```

Remarque 2.4 *La taille du vecteur extrait est celle du vecteur d'indices mais l'orientation (vecteur ligne ou colonne) est celle du vecteur source.*

On peut répliquer un vecteur plusieurs fois, pour avoir une matrice, en combinant l'opérateur `:` et un vecteur d'indices.

```
>> u=[1:3]'; v=u'
v =
   1  2  3

>> A=u(:,[1 1 1])
A =
   1     1     1
   2     2     2
   3     3     3


>> B=v([1 1],:)
B =
   1     2     3
   1     2     3
```

Pour extraire une sous-matrice, on peut aussi utiliser un vecteur d'indices ou une matrice d'indices pour spécifier la sous-matrice à extraire.

```
>> a=randn(5);           % matrice aléatoire d'ordre 5 (loi normale)
>> I=[2 4 3]; J=[2 1];   % indices des éléments à extraire
```

```
>> b=a(I,J)                % b(i,j)=a(I(i),J(j))
b =
   -0.5883   -0.0376
   -0.1364    0.1746
    2.1832    0.3273

>> IJ=[2 3; 2 3]           % matrice d'indices
IJ =
     2     3
     2     3

>> c=a(IJ)
c =
   -0.0376    0.3273
   -0.0376    0.3273
```

Dans le cas d'une matrice d'indices, l'indexation se fait comme suit, en supposant que la matrice d'indices `imat` est de taille 2×3

$$\mathtt{A(imat)} = \begin{bmatrix} \mathtt{A(imat(1,1)} & \mathtt{imat(1,2)} & \mathtt{imat(1,3)} \\ \mathtt{A(imat(2,1)} & \mathtt{imat(2,2)} & \mathtt{imat(2,3)} \end{bmatrix}$$

Comme dans un vecteur ou une matrice d'indices, un indice peut être répété, la matrice extraite peut-être théoriquement «plus grande» que la matrice source. Il s'agit alors d'une extension.

```
>> a=[1 4 7; 2 5 8; 3 6 9]'
a =
     1     2     3
     4     5     6
     7     8     9

>> IJ=[2 3 2 3; 1 2 1 2; 2 2 2 2; 1 1 1 1]  % les indices à extraire
IJ =
     2     3     2     3
     1     2     1     2
     2     2     2     2
     1     1     1     1

>> b=a(IJ)
b =
     4     7     4     7
     1     4     1     4
     4     4     4     4
     1     1     1     1

>> [m,n]=size(b)                            % taille de la matrice extraite
m =
   4
n =
   4
```

2.3.3 Suppression d'une ligne ou d'une colonne

Grâce à la matrice vide, `[]`, on peut supprimer une ligne ou une colonne d'une matrice.

```
>> A=[8 2 7; 0 1 3]
A =
   8     2     7
   0     1     3

>> A(:,2)=[]   % suppression de la 2ème colonne
A =
   8     7
   0     3

>> A(1,:)=[]   % suppression de la 1ère ligne
A =
   0     3
```

2.3.4 La fonction **diag**

La fonction `diag` permet de créer une matrice diagonale ou d'extraire la diagonale d'une matrice. Si `u` est un vecteur, alors `A=diag(u)` est la matrice diagonale dont la diagonale est `u`. Si `A` est une matrice, alors `v=diag(A)` est le vecteur contenant les éléments diagonaux de `A`. La fonction `diag(diag(A))` retourne une matrice diagonale dont la diagonale est celle de la matrice `A`.

```
>> A=hilb(4)+rand(4)
A =
    1.4565    1.1154    0.5096    0.6603
    0.5185    1.1253    0.6557    1.0936
    1.1547    1.1718    1.1355    0.2246
    0.6947    0.9382    1.0836    0.4957

>> u=diag(A)
u =
    1.4565
    1.1253
    1.1355
    0.4957

>> D=diag(u)
D =
    1.4565         0         0         0
         0    1.1253         0         0
         0         0    1.1355         0
         0         0         0    0.4957

>> DD=diag(diag(A))
DD =
    1.4565         0         0         0
```

```
         0    1.1253         0         0
         0         0    1.1355         0
         0         0         0    0.4957
```

La fonction `diag` admet un deuxième argument `k` permettant de désigner la $|k|$-ème sur-diagonale (si $k > 0$) ou sous-diagonale (si $k < 0$). Si A est une matrice carrée d'ordre n, alors `diag(A,k)` est un vecteur de taille $n - |k|$.

```
>> n=5;
>> H=pascal(5)
H =
   1     1     1     1     1
   1     2     3     4     5
   1     3     6    10    15
   1     4    10    20    35
   1     5    15    35    70

>> u=diag(H,2)
u =
     1
     4
    15
```

Si `u` est un vecteur de taille n, alors `diag(u,k)` est une matrice carrée, diagonale, d'ordre $n + |k|$.

```
>> n=4;
>> A=diag(2*ones(n-1,1),1)
A =
   0     2     0     0
   0     0     2     0
   0     0     0     2
   0     0     0     0
```

En combinant les différentes formes de la fonction `diag`, on peut générer une matrice tridiagonale. Comme exemple, voici comment générer la matrice de rigidité (à un coefficient près),

$$R = \begin{pmatrix} 2 & -1 & 0 & & \cdots & 0 \\ -1 & 2 & -1 & 0 & \cdots & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 \\ & & \ddots & \ddots & \ddots & \\ 0 & \cdots & 0 & -1 & 2 & -1 \\ 0 & \cdots & 0 & 0 & -1 & 2 \end{pmatrix}.$$

pour une triangulation uniforme sur un intervalle $]a,\ b[$.

```
>> n=5;
>> R=diag(2*ones(n,1))-diag(ones(n-1,1),1)-diag(ones(n-1,1),-1)
R =
   2    -1     0     0     0
```

```
  -1     2    -1     0     0
   0    -1     2    -1     0
   0     0    -1     2    -1
   0     0     0    -1     2
```

2.3.5 Matrices par blocs

La construction de matrices par blocs est particulièrement facile avec MATLAB. Soit à construire une matrice B de la forme

$$B = \begin{pmatrix} A & I \\ O & A \end{pmatrix}$$

où A est une matrice $n \times n$, O la matrice nulle d'ordre n et I_n la matrice identité d'ordre n.

```
>> A=pascal(3);                        % matrice de Pascal d'ordre 3
>> I3=eye(3);                          % matrice identité d'ordre 3
>> Zero=zeros(3);                      % matrice nulle d'odre 3
>> B=[A I3; Zero A]
B =

     1     1     1     1     0     0
     1     2     3     0     1     0
     1     3     6     0     0     1
     0     0     0     1     1     1
     0     0     0     1     2     3
     0     0     0     1     3     6
```

2.4 Normes vectorielles et matricielles

Les 3 normes vectorielles usuelles de $\mathbb{R}^n$, à savoir

$$\| x \|_1 = \sum_{i=1}^{n} |x_i|, \tag{2.1}$$

$$\| x \|_2 = \left(\sum_{i=1}^{n} x_i^2 \right)^{1/2}, \tag{2.2}$$

$$\| x \|_\infty = \max_{i=1,\dots,n} |x_i|, \tag{2.3}$$

son calculées respectivement avec `norm(x,1)`, `norm(x,2)` et `norm(x,inf)`. Par défaut, `norm(x)` retourne la norme Euclidienne de `x`.

```
>> x=rand(10,1);
>> n1=norm(x,1), n2=norm(x,2), ni=norm(x,inf)
n1 =
    5.6686
n2 =
```

```
    2.0042
ni =
    0.9501
```

Plus généralement, il est possible de calculer la norme p d'un vecteur de $\mathbb{R}^n$, à savoir

$$\| x \|_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p} .$$

Il suffit de le préciser dans le second argument de la fonction `norm`.

Lorsque le premier argument de la fonction `norm` est une matrice, la fonction `norm` retourne la norme matricielle subordonnée à la norme vectorielle spécifiée dans le deuxième argument de la fonction `norm`. La valeur par défaut pour le deuxième argument est toujours $p = 2$.

Soit A une matrice $m \times n$. La norme matricielle subordonnée à la norme p est donnée par

$$\| A \|_p = \max_{x \neq 0} \frac{\| Ax \|_p}{\| x \|_p}$$

Dans le cas des normes usuelles (i.e. $p = 1, 2, \infty$) on a.

$$\begin{aligned} \| A \|_1 &= \max_j \sum_i |a_{ij}|, \\ \| A \|_2 &= \left[\rho(A^T A) \right]^{1/2}, \\ \| A \|_\infty &= \max_i \sum_j |a_{ij}|, \end{aligned}$$

où $\rho(A^T A)$ est le rayon spectral de la matrice $A^T A$. Si A est une matrice carrée symétrique, alors

$$\| A \|_2 = \rho(A)$$

la plus grande valeur propre (en valeur absolue) de la matrice A.

```
>> A=rand(3,4)
A =
    0.4103    0.3529    0.1389    0.6038
    0.8936    0.8132    0.2028    0.2722
    0.0579    0.0099    0.1987    0.1988

>> n1=norm(A,1), n2=norm(A,2), ni=norm(A,inf)
n1 =
    1.3618
n2 =
    1.4520
ni =
    2.1818
```

La norme $\| \cdot \|_2$ étant difficile à calculer pour des matrices de grande taille (puisque faisant appel aux valeurs propres), on peut l'estimer avec la fonction `normest`.

La fonction `norm` peut aussi calculer la norme de Frobenius d'une matrice carrée, définie par

$$\| A \|_F = \left(\mathrm{tr}(A^T A)\right)^{1/2} = \left(\sum_{i,j=1}^{n} a_{ij}^2\right)^{1/2}$$

Pour cela, il suffit d'utiliser `norm(A,'fro')`.

```
>> a=hilb(4)
a =
    1.0000    0.5000    0.3333    0.2500
    0.5000    0.3333    0.2500    0.2000
    0.3333    0.2500    0.2000    0.1667
    0.2500    0.2000    0.1667    0.1429

>> nf=norm(a,'fro')
nf =
    1.5097
```

Fonction	Résultat
`abs`	valeur absolue ou module
`angle`	argument d'un nombre complexe
`imag`	partie imaginaire d'un nombre complexe
`real`	partie réelle d'un nombre complexe
`conj`	nombre complexe conjugué

TABLE 2.3 – Quelques fonctions mathématiques élémentaires complexes

2.5 Fonction mathématiques élémentaires

Les fonctions mathématiques élémentaires sont définies à la fois pour des scalaires (réels ou complexes) et pour les tableaux. Dans le cas des tableaux, le résultat est un tableau de même dimension que le tableau passé en argument. La fonction s'applique sur chaque élément du tableau. Le tableau 2.3-2.5 donne quelques fonctions mathématiques élémentaires usuelles. Pour la liste complète, taper `help elfun`.

2.6 Exercices

Exercice 2.1 Tabuler, dans l'intervalle $[-1,\ 1]$, la fonction

$$f(x) = \begin{cases} \frac{1}{x}\sin x & \text{si } x \neq 0, \\ 1 & \text{si } x = 0 \end{cases}$$

Fonction	Résultat
`cos`	cosinus
`acos`	arc cosinus
`cosh`	cosinus hyperbolique
`sin`	sinus
`asin`	arc sinus
`sinh`	sinus hyperbolique
`tan`	tangente
`atan`	arc tangente
`tanh`	tangente hyperbolique

TABLE 2.4 – Quelques fonctions trigonométriques

en utilisant seulement les opérateurs vus dans ce chapitre en évitant l'apparition de `nan` à cause d'une division par zéro, tout en incluant zéro dans la tabulation de l'intervalle. On pourra représenter f à l'aide de la fonction `plot(x,y)` qui trace la courbe de $y = f(x)$.

Fonction	Résultat
`exp`	exponentielle
`log`	logarithme népérien
`log10`	logarithme base 10
`sqrt`	racine carrée
`nthroot`	racine n-ème

TABLE 2.5 – Exponentielles et logarithmes

Exercice 2.2 (Polynômes de Tchebychev) On appelle polynôme de Tchebychev de degré n la fonction $T_n : [-1, 1] \to \mathbb{R}$, définie par

$$T_n(x) = \cos(n \arccos x).$$

Tabuler les fonctions T_n, pour $0 \leq n \leq 4$, dans le même tableau. On n'utilisera qu'une seule instruction MATLAB pour l'évaluation de toutes les fonctions T_n.

Exercice 2.3 Soit U une matrice $n \times 3$, dont chaque ligne représente un vecteur de $\mathbb{R}^3$. Écrire une instruction MATLAB qui calcule la norme euclidienne de toutes les lignes de la matrice U.

Exercice 2.4 Soit x et y deux vecteurs de $\mathbb{R}^n$. Écrire les instructions MATLAB pour générer les matrices

suivantes

$$\begin{aligned} a_{ij} &= x_i y_j \\ b_{ij} &= x_i + y_j \\ c_{ij} &= \frac{x_i}{y_j} \\ d_{ij} &= x_i^{|y_j|} \end{aligned}$$

pour $i, j = 1, \ldots, n$.

Exercice 2.5 Soit $K = (k_{ij})$ une matrice carrée d'ordre n. Construire une matrice carrée $A = (a_{ij})$ d'ordre $3n$, dont tous les éléments sont nuls sauf

$$a_{3i-2,3j-2} \leftarrow k_{ij}, \quad a_{3i-1,3j-1} \leftarrow k_{ij}, \quad a_{3i,3j} \leftarrow k_{ij}, \qquad i, j = 1, \ldots, n.$$

Exercice 2.6 (Matrice de Vandermonde) Pour $x = (x_1 \; x_2 \; \cdots \; x_n)^T$ un vecteur de $\mathbb{R}^n$, une matrice de Vandermonde est de la forme

$$V = \begin{bmatrix} x_1^m & x_1^{m-1} & \cdots & x_1 & 1 \\ x_2^m & x_2^{m-1} & \cdots & x_2 & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_n^m & x_n^{m-1} & \cdots & x_n & 1 \end{bmatrix}$$

Générer la matrice V.

CHAPITRE 3

ALGÈBRE LINÉAIRE

Toute la puissance de MATLAB est symbolisée par l'opérateur \ utilisé pour la résolution des systèmes linéaires quelque soit leur nature (triangulaire, carré, rectangulaire sur ou sous déterminé). Plusieurs fonctions de factorisation de matrices sont également disponibles.

3.1 Systèmes linéaires

L'opérateur MATLAB \ (*backslash*) est la commande générique pour résoudre un système d'équations linéaires

$$Ax = b \tag{3.1}$$

avec A une matrice $m \times n$ et $b \in \mathbb{R}^n$. Avant le calcul de la solution, les coefficients de la matrice A sont examinés. Ainsi l'opérateur \ détecte automatiquement les cas suivants:

— A est triangulaire (moyennant quelques permutations);
— A est symétrique définie positive
— A est carrée non singulière n'entrant pas dans les deux cas précédents;
— A est rectangulaire surdéterminée ($m > n$) ou surdéterminée ($m < n$).

3.1.1 Systèmes carrés

C'est le cas le plus courant: A est une matrice carrée d'ordre n et b un vecteur de $\mathbb{R}^n$.

```
>> A=hilb(4)       % matrice du système
A =
    1.0000    0.5000    0.3333    0.2500
    0.5000    0.3333    0.2500    0.2000
    0.3333    0.2500    0.2000    0.1667
    0.2500    0.2000    0.1667    0.1429

>> b=ones(4,1)     % second membre du système
b =
   1
   1
   1
   1
```

```
>> x=A\b           % resolution
x =
   -4.0000
   60.0000
 -180.0000
  140.0000
```

Lorsque la matrice A est singulière ou presque singulière un message d'avertissement est émis. Dans l'exemple suivant A est une matrice de Hilbert d'ordre 4 rendue presque singulière (la première et la quatrième colonnes sont presque identiques). La norme du résidu indique que la solution calculée est loin de la solution exacte.

```
>> A=hilb(4); A(:,4)=A(:,1)+10^-15;
>> b=ones(4,1);
>> x=A\b
Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = 7.103259e-18.
x =
   1.0e+15 *

   -1.1262
    0.0000
   -0.0000
    1.1262

>> norm(A*x-b)
ans =
    0.0021
```

Si on a plusieurs seconds membres $b_1, \ldots, b_m$ disponibles, on en fait une matrice second membre $B = (b_1 \cdots b_m)$ et on applique l'opérateur \ avec cette matrice. On gagne du temps car les colonnes de B sont transformées simultanément pendant l'élimination. La solution $X = (x_1 \cdots x_m)$ est alors une matrice dont chaque colonne x_i est la solution du système $Ax = b_i$.

```
>> A=hilb(4)
>> b1=ones(4,1), b2=rand(4,1), b3=randn(4,1)
>> B=[b1 b2 b3]
B =
    1.0000    0.9501   -0.4326
    1.0000    0.2311   -1.6656
    1.0000    0.6068    0.1253
    1.0000    0.4860    0.2877

>> x=A\B
x =
  -4.0000e+00   6.5070e+01   1.8275e+02
   6.0000e+01  -6.5867e+02  -1.8019e+03
  -1.8000e+02   1.4952e+03   3.9972e+03
   1.4000e+02  -9.3269e+02  -2.4585e+03
```

3.1.2 Systèmes rectangulaires

Les systèmes rectangulaires surdéterminés ($m > n$) se rencontrent le plus souvent en identification de paramètres. Si A est une matrice $m \times n$, alors le système (3.1) admet une solution si $b \in \mathrm{Im}A$. Ce qui est peu probable si $m >> n$. Dans ce cas on peut se contenter d'une projection orthogonale de b sur $\mathrm{Im}A$. Ce qui revient à résoudre le problème de moindres carrés

$$\min_{x \in \mathbb{R}^n} \frac{1}{2} \parallel Ax - b \parallel_2^2 . \tag{3.2}$$

Une solution de (3.2) vérifie le système aux équations normales

$$A^T Ax = A^T b. \tag{3.3}$$

Ce système a toujours au moins une solution. Si A est de rang plein, alors $A^T A$ est inversible et l'unique solution de (3.3) est donnée par

$$x = (A^T A)^{-1} A^T b.$$

La matrice $A^+ = (A^T A)^{-1} A^T$ est appelée *pseudo-inverse* de A car $A^+ A = \mathbb{I}_n$ (mais $AA^+ \neq \mathbb{I}$).

t	y
-1.00	4.00
-0.75	5.25
-0.50	3.00
0.00	5.00
0.30	6.75
0.50	7.00
1.00	9.00

TABLE 3.1 – Tableau des observation du modèle linéaire

Un exemple simple est la régression linéaire. Un modèle linéaire à une entrée $y(t) = a + tb$ est mesuré en différentes valeurs de t pour produire les observations du tableau 3.1. Les paramètres a et b sont à déterminer pour minimiser (au sens des moindres carrés) l'erreur des observations du tableau 3.1.

Cela donne le système surdéterminé

$$\begin{aligned}
a - 1.00b &= 4.00 \\
a - 0.75b &= 5.25 \\
a - 0.50b &= 3.00 \\
a + 0.00b &= 5.00 \\
a + 0.30b &= 6.75 \\
a + 0.50b &= 7.00 \\
a + 1.00b &= 9.00
\end{aligned}$$

La solution aux moindres carrés de ce système est calculé avec l'opérateur $\backslash$.

```
>> A=[1 -1; 1 -0.75; 1 -0.5; 1 0; 1 0.3; 1 0.5 1 1]
A =
   1.0000   -1.0000
   1.0000   -0.7500
   1.0000   -0.5000
   1.0000         0
   1.0000    0.3000
   1.0000    0.5000
   1.0000    1.0000

>> b=[4 5.25 3 5 6.75 7 9]
b =
   4.0000
   5.2500
   3.0000
   5.0000
   6.7500
   7.0000
   9.0000

>> x=A\b
X =
   5.8719
   2.4520
```

La droite de régression $y = 5.8719 + 2.4520t$ et les points de l'échantillon sont représentées dans la figure 3.1

Les systèmes surdéterminés se rencontrent aussi dans les problèmes d'éléments finis lorsque les conditions aux bords sont des relations linéaires entre variables. Dans ce cas, le plus simple est de rajouter ces relations aux système d'équations fournies par la discrétisation. On obtient alors un système surdéterminé.

Pour un système sous-déterminé ($m < n$), il y a plus d'inconnues que d'équations. La matrice $A^T A$ du système aux équations normales (3.3) est donc singulière. En conséquence, la solution n'est jamais unique. MATLAB fourni une solution particulière (en utilisant la factorisation QR).

```
>> A=pascal(4); A(4,:)=[]   % suppression de la 4ème ligne de A
A =
   1     1     1     1
   1     2     3     4
   1     3     6    10

>> b=[2;3;7];
>> x=A\b
x =
   2.3333
   0
  -2.0000
   1.6667
```

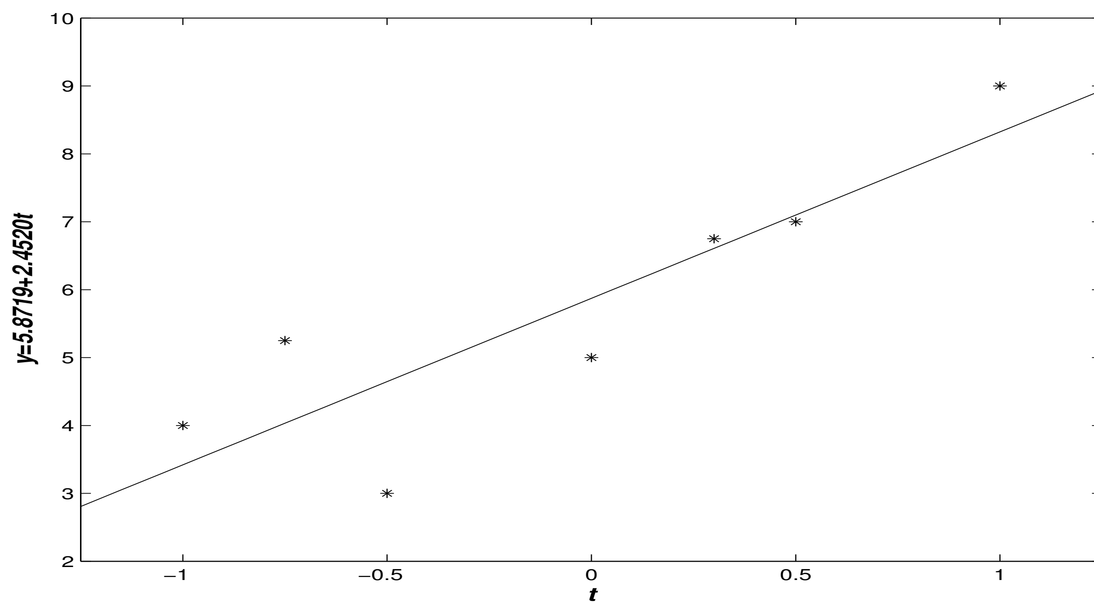

FIGURE 3.1 – Régression linéaire

```
>> norm(A*x-b)
ans =
   2.1756e-15
```

Une solution générale est la somme de la solution particulière et d'un vecteur du noyau dont la base est déterminée à l'aide de la fonction `null` (*cf.* l'aide en ligne).

3.2 Inverses et déterminants

Les fonctions `det` et `inv` calculent le déterminant et la matrice inverse d'une matrice carré passée en argument.

```
>> a=hilb(4);
>> det(a)
ans =
   1.6534e-07

>> ai=inv(a)
ai =
   1.6000e+01  -1.2000e+02   2.4000e+02  -1.4000e+02
  -1.2000e+02   1.2000e+03  -2.7000e+03   1.6800e+03
   2.4000e+02  -2.7000e+03   6.4800e+03  -4.2000e+03
  -1.4000e+02   1.6800e+03  -4.2000e+03   2.8000e+03
```

```
>> ai*a                                              % vérification
ans =

   1.0000e+00            0   3.5527e-15            0
            0   1.0000e-00            0  -2.8422e-14
            0   1.1369e-13   1.0000e+00  -1.1369e-13
            0  -1.1369e-13  -5.6843e-14   1.0000e+00
```

La matrice de Pascal a une inverse entière.

```
>> P=pascal(5)
P =

     1     1     1     1     1
     1     2     3     4     5
     1     3     6    10    15
     1     4    10    20    35
     1     5    15    35    70

>> Pi=inv(P)
Pi =

     5   -10    10    -5     1
   -10    30   -35    19    -4
    10   -35    46   -27     6
    -5    19   -27    17    -4
     1    -4     6    -4     1
```

Pour résoudre le système (3.1), lorsque A est une matrice non singulière, on a donc (théoriquement) la possibilité d'utiliser l'une des deux instructions suivantes.

```
>> x=A\b            % élimination de Gauss
>> x=inv(A)*b       % inversion de A
```

Mais pour des raisons de coûts et de stabilité numérique, la première est de loin préférable.

Pour une matrice rectangulaire $m \times n$ ($m > n$), de rang plein, la matrice pseudo-inverse $A^+ = (A^T A)^{-1} A^T$, est calculée par la fonction `pinv`.

```
>> A=randn(4,2)
A =
    0.6145   -0.6436
    0.5077    0.3803
    1.6924   -1.0091
    0.5913   -0.0195

>> Api=pinv(A)
Api =
   -0.1124    0.6438    0.3071    0.3763
   -0.5449    1.0254   -0.2656    0.4460
```

```
>> Api*A                    % Verification : pinv(A)*A=I
ans =
    1.0000    0.0000
    0.0000    1.0000

>> A*Api                    % Verification : A*pinv(A) <> I
ans =
    0.2816   -0.2643    0.3596   -0.0558
   -0.2643    0.7169    0.0549    0.3607
    0.3596    0.0549    0.7877    0.1867
   -0.0558    0.3607    0.1867    0.2138
```

A noter que lorsque la matrice du système (3.1) est rectangulaire, `x=A\b` donne la solution aux moindres carrés. Théoriquement, les trois instructions suivantes calculent la solution du système aux équations normales (3.3).

```
>> x=A\b
>> x=pinv(A)*b
>> x=inv(A'*A)*A'*b
```

Comme pour les systèmes non singuliers, seule la première est utilisée pour des raisons de coûts et de stabilité numérique.

3.3 Valeurs propres, vecteurs propres

La fonction `eig` retourne les valeurs propres d'une matrice carrée passée en argument.

```
>> A=pascal(4)
A =
     1     1     1     1
     1     2     3     4
     1     3     6    10
     1     4    10    20

>> lambda=eig(A)
lambda =
   3.8016e-02
   4.5383e-01
   2.2034e+00
   2.6305e+01
```

Si de plus A est diagonalisable (*i.e.* les vecteurs propres de A forment une base de $\mathbb{R}^n$), on peut obtenir les vecteurs propres associés en utilisant deux arguments résultat

```
[P,D]=eig(A)
```

où `P` est la matrice de passage (vecteurs propres normés stockés en colonne) et `D` la matrice diagonale des valeurs propres. Avec la matrice A de l'exemple précédent on a.

```
>> [P,D]=eig(A)

P =

   3.0869e-01  -7.8728e-01   5.3037e-01   6.0187e-02
  -7.2309e-01   1.6323e-01   6.4033e-01   2.0117e-01
   5.9455e-01   5.3211e-01   3.9183e-01   4.5808e-01
  -1.6841e-01  -2.6536e-01  -3.9390e-01   8.6375e-01


D =

   3.8016e-02            0            0            0
            0   4.5383e-01            0            0
            0            0   2.2034e+00            0
            0            0            0   2.6305e+01

>> P*D*inv(P)                                    % vérification A=P*D*inv(P)

ans =

    1.0000    1.0000    1.0000    1.0000
    1.0000    2.0000    3.0000    4.0000
    1.0000    3.0000    6.0000   10.0000
    1.0000    4.0000   10.0000   20.0000
```

3.4 Factorisations de matrices

Fonction	factorisation
`lu`	factorisation LU
`chol`	factorisation de Cholesky
`qr`	factorisation QR
`luinc`	factorisation LU incomplète
`cholinc`	factorisation de Cholesky incomplètes

TABLE 3.2 – Quelques fonctions de factorisation de matrices

Quelques fonctions MATLAB de factorisation de matrices sont présentées figure 3.2. Les fonctions `lu` et `chol` sont surtout utiles lorsqu'on doit résoudre plusieurs systèmes avec la même matrice. Dans ce cas, après factorisation, les différentes résolutions se réduisent à des substitutions. D'où un gain en temps de calcul. Les fonctions de factorisation incomplète (`luinc` et `cholinc`) seront étudiées au chapitre 5. Elles servent à calculer les préconditionneurs des méthodes itératives. Pour voir toutes les fonctions MATLAB de factorisation ou de décomposition de matrices, taper `lookfor factorization` ou `lookfor decomposition` dans le fenêtre de commandes.

3.4.1 Factorisation LU

Si A une matrice carrée d'ordre n non singulière, alors il existe une matrice L, triangulaire inférieure, et une matrice U triangulaire supérieure telle que $A = LU$. La solution du système (3.1) s'obtient alors en résolvant successivement

$$\begin{aligned} Ly &= b, \\ Ux &= y. \end{aligned}$$

La forme la plus simple de la fonction `lu` a pour syntaxe

```
[L,U]=lu(A);
```

Ce qui donne, pour la résolution,

```
y=L\b;
x=U\y;
```

ou encore

```
x=U\(L\b);
```

Si la stratégie du pivot partiel est appliquée lors de la factorisation, alors il existe, en plus de L et U, une matrice de permutation P telle que $PA = LU$. La syntaxe de la fonction `lu` devient alors

```
[L,U,P]=lu(A)
```

Il faut alors permuter le second membre lors de la résolution, *i.e.* remplacer (3.1) par le système

$$PAx = Pb$$

Voici un exemple complet.

```
>> A=randn(4)                % A générée aléatoirement
A =
    0.2944   -0.6918   -1.4410    0.8156
   -1.3362    0.8580    0.5711    0.7119
    0.7143    1.2540   -0.3999    1.2902
    1.6236   -1.5937    0.6900    0.6686

>> [L,U]=lu(A)               % factorisation de A
L =
    0.1813   -0.2060    1.0000         0
   -0.8230   -0.2320   -0.5703    1.0000
    0.4400    1.0000         0         0
    1.0000         0         0         0

U =
    1.6236   -1.5937    0.6900    0.6686
         0    1.9552   -0.7035    0.9961
```

```
         0         0   -1.7110    0.8996
         0         0         0    2.0063

>> b=ones(4,1)               % vecteur ne contenant que des 1
>> x=U\(L\b)                 % résolution de Ly=b et Ux=y
x =
   -0.2402
   -0.3068
    0.1055
    1.2389
```

On remarque surtout que la matrice L n'est pas triangulaire inférieure car la stratégie du pivot partiel est systématiquement appliquée dans la fonction `lu`. En fait L est le produit d'une matrice de permutation et d'une matrice triangulaire inférieure. Pour avoir la conscience tranquille, il faut donc «isoler» les permutations dans une matrice à part. Avec la matrice `A` de l'exemple précédent on a

```
>> [L,U,P]=lu(A)
L =

    1.0000         0         0         0
    0.4400    1.0000         0         0
    0.1813   -0.2060    1.0000         0
   -0.8230   -0.2320   -0.5703    1.0000

U =
    1.6236   -1.5937    0.6900    0.6686
         0    1.9552   -0.7035    0.9961
         0         0   -1.7110    0.8996
         0         0         0    2.0063

P =
     0     0     0     1
     0     0     1     0
     1     0     0     0
     0     1     0     0

>> x=U\(L\(P*b))       % résolution avec permutation du 2nd membre
x =
   -0.2402
   -0.3068
    0.1055
    1.2389
```

Remarque 3.1 *Il existe un deuxième argument permettant de contrôler la recherche du pivot dans* `lu`*, voir § 5.4.1.*

3.4.2 Factorisation de Cholesky

Si la matrice A est symétrique définie positive, alors il existe une matrice unique R, triangulaire inférieure ayant tous les éléments diagonaux positifs, telle que

$$A = RR^T.$$

C'est la factorisation de Cholesky qui s'obtient par la fonction `chol`. Le système (3.1) est alors remplacé par

$$RR^T x = b.$$

Mais la fonction `chol` retourne R^T de sorte que la factorisation de Cholesky de MATLAB est `A=R'*R` au lieu de $A = RR^T$. En tenant compte de ce détail, la solution du système est donné par

```
y=R'\b;
x=R\y;
```

ou encore

```
x=R\(R'\b)
```

La matrice de Pascal, donnée par la fonction `pascal` est définie positive.

```
>> A=pascal(5)                                          % matrice de Pascal d'ordre 5
A =
     1     1     1     1     1
     1     2     3     4     5
     1     3     6    10    15
     1     4    10    20    35
     1     5    15    35    70

>> R=chol(A)                                            % factorisation de Cholesky
R =
     1     1     1     1     1
     0     1     2     3     4
     0     0     1     3     6
     0     0     0     1     4
     0     0     0     0     1

>> b=randperm(5)'                                       % second membre
b =
     3
     5
     2
     1
     4

>> x=R\(R'\b)                                           % résolution
x =
   -16
    53
```

```
   -56
    27
    -5

>> norm(A*x-b)                                     % vérification
ans =
     0
```

Avant d'appeler la fonction `chol` il est important de s'assurer que la matrice passée en argument est définie positive.

```
>> A=rand(3);     % matrice quelconque
>> R=chol(A)      % tentative de factorisation R'R
??? Error using ==> chol
Matrix must be positive definite.
```

3.4.3 Factorisation QR

Une matrice Q, $m \times n$ ($m \geq n$), est dite orthogonale ou unitaire si ses colonnes sont des vecteurs orthogonaux deux à deux et de norme unité. Si Q est une matrice orthogonale, on a

$$Q^T Q = \mathbb{I}_n.$$

Pour toute matrice A, $m \times n$, il existe une matrice orthogonale Q, $m \times n$, et une matrice rectangulaire supérieure R, $n \times n$, telles que

$$A = QR$$

La fonction `qr` retourne les facteurs Q et R de la factorisation.

```
>> P=pascal(4); A=P(:,1:2)
A =
   1     1
   1     2
   1     3
   1     4

>> [Q,R]=qr(A)
Q =
   -0.5000    0.6708    0.0236    0.5472
   -0.5000    0.2236   -0.4393   -0.7120
   -0.5000   -0.2236    0.8079   -0.2176
   -0.5000   -0.6708   -0.3921    0.3824

R =
   -2.0000   -5.0000
         0   -2.2361
         0         0
         0         0

>> Q*R
```

```
ans =
    1.0000    1.0000
    1.0000    2.0000
    1.0000    3.0000
    1.0000    4.0000
```

On remarque surtout que la matrice Q est carrée d'ordre 4 au lieu d'être une matrice 4×2. Par défaut les colonnes de A sont complétées par des colonnes linéairement indépendantes pour avoir une matrice Q carrée. Dans beaucoup de cas, on a pas besoin de la matrice Q complète, car les colonnes rajoutées sont dans l'orthogonal de $\mathrm{Im}A$ et correspondent à la partie nulle de la matrice R. Dans ce cas, on utilise la forme économique de la fonction `qr` qui permet de ne retenir que les partie utile de la matrice Q.

```
>> [Q,R]=qr(A,0)
Q =
   -0.5000    0.6708
   -0.5000    0.2236
   -0.5000   -0.2236
   -0.5000   -0.6708

R =
   -2.0000   -5.0000
         0   -2.2361

>> Q*R
ans =
    1.0000    1.0000
    1.0000    2.0000
    1.0000    3.0000
    1.0000    4.0000
```

3.4.4 Décomposition en valeurs singulières

Pour une matrice rectangulaire A la notion de valeur propre n'est pas valide. On introduit le concept de valeur singulière. Les valeurs singulières μ_i d'une matrice rectangulaire A sont les racines carrées des valeurs propres de la matrice carrée $A^T A$.

Si A est une matrice rectangulaire $m \times n$, il existe deux matrices orthogonales U et V, d'ordre respectivement m et n telles que

$$A = USV^T$$

où S est la matrice diagonale des valeurs singulières. C'est la décomposition en valeurs singulières fournie par la fonction `svd`.

```
>> P=pascal(4); A=P(:,2:3)
A =
   1     1
   2     3
   3     6
   4    10
```

```
>> [U,S,V]=svd(A)
U =
   -0.0999   -0.5355   -0.5459   -0.6366
   -0.2687   -0.6366   -0.1508    0.7070
   -0.5065   -0.3031    0.7469   -0.3061
   -0.8132    0.4649   -0.3483    0.0352

S =
   13.2331         0
         0    0.9408
         0         0
         0         0

V =
   -0.4088   -0.9126
   -0.9126    0.4088

>> U*S*V'                                          % vérification
ans =
    1.0000    1.0000
    2.0000    3.0000
    3.0000    6.0000
    4.0000   10.0000
```

3.5 Exercices

Exercice 3.1 Considérons la matrice tridiagonale d'ordre n suivante, étudiée au § 2.3.4.

$$R = \begin{pmatrix} 2 & -1 & 0 & & \cdots & 0 \\ -1 & 2 & -1 & 0 & \cdots & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 \\ & & \ddots & \ddots & \ddots & \\ 0 & \cdots & 0 & -1 & 2 & -1 \\ 0 & \cdots & 0 & 0 & -1 & 2 \end{pmatrix}.$$

Sa condition $\kappa_2(R)$ est de l'ordre de n^2 ($O(n^2)$), voir par exemple [11, chap. 2] ou [7, chap. 4] pour la formule exacte. Il existe donc trois réels a, b et c tels que $\kappa_2(R) = an^2 + bn + c$. On se propose de calculer les valeurs approchées de a, b et c par la méthode des moindres carrés. Calculer $\kappa_2(R)$ pour $n = 5, 10, 30, 50, 100$. Former la matrice et le second membre du système surdéterminé correspondant. Calculer les valeurs approchées de a, b et c au sens des moindres carrés.

Exercice 3.2 Un modèle quadratique à une entrée $y(t) = x_0 + x_1 t + x_2 t^2$ est mesuré en différentes valeurs de t pour produire les observations du tableau 3.3. Les paramètres x_0, x_1 et x_2 sont à déterminer. Générer la matrice A et le second membre b du système surdéterminé correspondant et calculer les paramètres x_0, x_1 et x_2. Tracer (avec `plot`) la courbe $y(t)$, $t \in [-5,\ 5]$ et les points du tableau 3.3.

t	-4	-3	-1.5	-0.5	1	2.5	4
y	6	3	-2	-1	2	10	23

TABLE 3.3 – Tableau des observations du modèle quadratique de l'exercice 3.2

Exercice 3.3 Cet exercice [11] montre que la valeur du déterminant n'est pas une indication sur le bon ou mauvais conditionnement d'une matrice. Soit la matrice carrée d'ordre n

$$A = \begin{pmatrix} 1 & 2 & 0 & & \cdots & 0 \\ 0 & 1 & 2 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 2 & 0 & 0 \\ & & \ddots & \ddots & \ddots & \\ 0 & \cdots & 0 & 0 & 1 & 2 \\ 0 & \cdots & 0 & 0 & 0 & 1 \end{pmatrix}.$$

Générer la matrice A (pour des valeurs de n raisonnables). Calculer le déterminant, l'inverse et la condition $\kappa(A)$ avec les normes $\|\cdot\|_1$ ou $\|\cdot\|_\infty$. Tracer la courbe de $\kappa_1(A)$ en fonction de n.

Exercice 3.4 On considère la matrice

$$A = \begin{pmatrix} -1 & 2 & 2 \\ -3 & 4 & 2 \\ -1 & 1 & 3 \end{pmatrix}.$$

1.– Calculer les valeurs et vecteurs propres de A, *i.e.* une matrice diagonale D et une matrice inversible P telle que $A = PDP^{-1}$.

2.– On considère la suite vectorielle

$$\begin{aligned} u_n &= -u_{n-1} + 2v_{n-1} + 2w_{n-1}, \\ v_n &= -3u_{n-1} + 4v_{n-1} + 2w_{n-1}, \\ w_n &= -u_{n-1} + v_{n-1} + 3w_{n-1}, \end{aligned}$$

pour $n \geq 1$ et $u_0 = v_0 = w_0 = 1$. Calculer u_{10}, v_{10} et w_{10}.

Exercice 3.5 Soit la matrice triadiagonale d'ordre n

$$A = \begin{pmatrix} 2+2h^2 & -1-2h^2 & 0 & & \\ -1-2h^2 & 2+8h^2 & -1-6h^2 & & \\ & \ddots & \ddots & & \\ & & & 2+2(n-1)^2h^2 & -1-n(n-1)h^2 \\ & & & -1-n(n-1)h^2 & 2+2n^2h^2 \end{pmatrix}$$

où $h = 1/(n+1)$, et le vecteur de $\mathbb{R}^n$

$$b = -h^2 \left(f_1 - \frac{1}{h^2}, f_2, \ldots, f_{n-1}, f_n - [n(n+1) + 1/h^2](2 + \sin 1) \right)^T$$

où $f_i = 2 + 6x_i^2 + 2x_i \cos x_i - (1 + x_i^2) \sin x_i$ avec $x_i = ih$, $i = 1, \ldots, n$.

La matrice A et le vecteur b sont issus d'une discrétisation par différences finies du problème aux limites [7]

$$(1+x^2)u''(x) + 2xu'(x) = 2 + 6x^2 + 2x\cos x - (1+x^2)\sin x \quad \text{dans }]0,\ 1[,$$
$$u(0) = 1, \quad u(1) = 2 + \sin 1$$

dont la solution exacte est $u_e(x) = x^2 + \sin x + 1$.

1.– Générer la matrice A et le second membre b pour n donné, $n \leq 100$.

2.– Résoudre le système $Au_h = b$, où $u_h = (u_1, \ldots, u_n)^T$ est la solution approchée (aux points $x_i = ih$) du problème aux limites.

3.– Calculer l'erreur, en norme L^2, entre la solution approchée u_h et la solution exacte u_e

$$\| u_h - u_e \|_{L^2(]0,\,1[)} = \int_0^1 (u_h(x) - u_e(x))^2 \,\mathrm{d}x \approx \left(h \sum_{i=1}^{n} (u_i - u_e(x_i))^2 \right)^{1/2}.$$

CHAPITRE 4

GRAPHISME

En plus du calcul pur, MATLAB est réputé pour son interface graphique. Vecteurs et matrices peuvent être visualisés sous forme de courbes, surfaces, courbes de niveau, histogrammes, etc. Ces graphiques peuvent êtres annoter, légender et modifier à partir de la ligne de commande ou directement dans la fenêtre graphique.

Pour l'aide en ligne sur les fonctions graphiques

help graph2d graphes bidimensionnels;

help graph3d graphes tridimensionnels;

help specgraph graphes spéciaux (barres, histogrammes, camemberts, etc).

4.1 Généralités

En MATLAB une instruction graphique ouvre une fenêtre graphique dans laquelle est affichée le résultat. La commande `figure` permet d'ouvrir une nouvelle fenêtre. Par défaut, c'est la dernière fenêtre ouverte qui est active (donc visible). Pour réactiver une fenêtre devenue inactive, il suffit de taper `figure(n)`, où `n` est le numéro de la figure. Si la figure `n` n'existe pas, une nouvelle fenêtre est créée avec comme numéro le numéro suivant celui de la dernière fenêtre ouverte.

4.2 Graphisme 2D

La fonction de base pour le graphisme 2D est `plot`. La forme simple de la fonction `plot` est

```
plot(x,y)
```

trace la courbe définie par l'ensemble des points (x_i, y_i), $i = 1, .., n$. Les deux vecteurs `x` et `y` doivent donc être de même taille. Le tracé est obtenu en reliant les points (x_i, y_i) par des segments de droite.

```
>> x=-4:.1:4;
>> y=x.*sin(x).*cos(x);
>> plot(x,y)
```

Le résultat est la figure 4.1.

Lorsqu'on trace plusieurs courbes sur la même fenêtre, on aimerait pouvoir les distinguer les unes des autres. La commande `plot` dispose d'un troisième paramètre d'entrée permettant de spécifier la couleur,

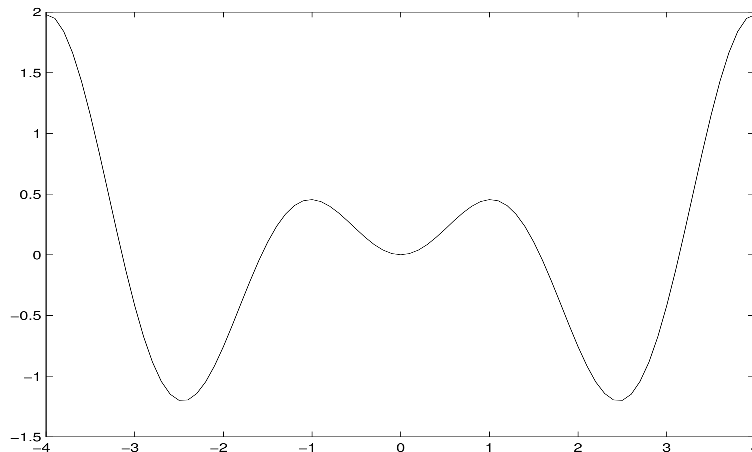

FIGURE 4.1 – Graphe de la fonction $y = x \sin x \cos x$ entre -4 et 4.

le symbole à chaque point (x_i, y_i), le type de tracé. Ce troisième paramètre est une chaîne d'au plus 3 caractères de la forme `'cst'`, où `c` est la couleur de la courbe, `s` le symbole à chaque point (x_i, y_i) et `t` le style du trait. Les différentes possibilités offertes par MATLAB sont présentées dans le tableau 4.1.

Avec la possibilité de distinguer les tracés, on peut utiliser la forme complète de la commande `plot`. On utilise alors la syntaxe

```
plot(x1,y1,s1,x2,y2,s2,...)
```

qui permet de tracer (dans le même figure) les courbes définies par les vecteurs de points `(x1,y1)`, `(x2,y2)`,... avec `s1`, `s2`,... les spécifications permettant de les distinguer.

Voici un exemple d'utilisation de la commande `plot` complète. Le résultat est la figure 4.2

```
>> x=-4:0.1:4; t=-4:.5:4;
>> y1=x.*sin(x).*cos(x); y2=exp(-x.*x+sin(x)); z=exp(sin(t));
>> plot(x,y1,'k-',x,y2,'r:',t,z,'*')
```

Pour le troisième tracé, *i.e.* `(t,z)`, on a juste marqué les points (t_i, z_i) par le symbole `*`.

Si les différentes courbes doivent être tracées l'une après l'autre, il faut maintenir la fenêtre précédente avec la commande `hold on`. Sinon, la nouvelle fenêtre efface l'ancienne (et donc la courbe précédente). Les différentes courbes de la figure 4.2 peuvent être tracées l'une après l'autre, comme suit.

```
>> x=-4:0.1:4; t=-4:.5:4;
>> y1=x.*sin(x).*cos(x);
>> plot(x,y1,'k-')
>> y2=exp(-x.*x+sin(x));
>> hold on
>> plot(x,y2,'r:')
>> z=exp(sin(t));
>> plot(t,z,'*')
>> hold off
```

Couleur		Symbole		Style	
y	jaune	.	point	−	trait plein
m	magenta	o	rond	:	pointillé court
c	cyan	x	croix	−.	pointillé long
r	rouge	+	plus	−-	pointillé mixte
g	vert	⋆	étoile		
b	bleu	s	carré		
w	blanc	d	losange		
k	noir	v	triangle (bas)		
		ˆ	triangle (haut)		
		<	triangle (gauche)		
		>	triangle (doit)		
		p	pentagone		
		h	hexagone		

TABLE 4.1 – Couleur, symbole et style d'un tracé

Avec la commande `hold off` toute nouvelle figure efface la précédente. C'est le mode par défaut. Lorsqu'on tape seulement `hold` on passe d'un mode à un autre, *i.e.* on passe de `hold on` à `hold off` et *vice versa*.

4.2.1 Les axes

La fonction `axis` permet de régler les axes. La syntaxe est

```
axis([x_min x_max y_min y_max])
```

qui fixe les bornes min et max des axes. Par défaut, une figure comporte systématiquement des axes. Pour les supprimer, il suffit de taper `axis off`; et pour les rétablir `axis on`. Dans tous les cas, pour passer d'un état à un autre on tape simplement `axis`. Lorsqu'on souhaite que la figure respecte absolument l'échelle du graphe, il suffit de taper `axis image`. Pour resserrer les axes autour d'une figure, sans forcement tenir compte de l'échelle, la commande est `axis tight`.

Les commandes `xlabel` et `ylabel` permettent de légender les axes du graphe. La syntaxe est

```
xlabel('texte')
ylabel('texte')
```

Pour les textes comportant des formules mathématiques (*e.g.* x_1) ou des lettres grecques, on utilise le codage Latex (voir, par exemple,[9, 10]). Le code suivant donne la figure 4.3.

```
>> theta=-2:.1:2; y=exp(sin(theta));
>> plot(theta,y)
>> xlabel('\theta')
```

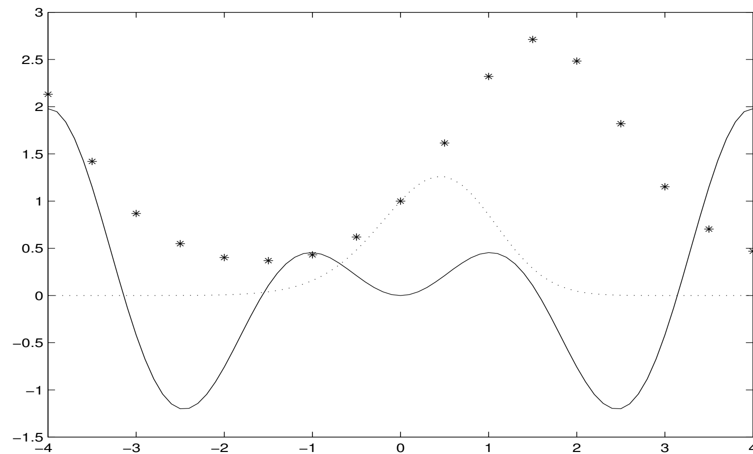

FIGURE 4.2 – Graphes des fonctions $y = x \sin x \cos x$ (trait plein), $y = \mathrm{e}^{-x^2+\sin x}$ (pointillé) et $y = \mathrm{e}^{\sin x}$ ($\star$)

```
>> ylabel('y=e^{sin\theta}')
```

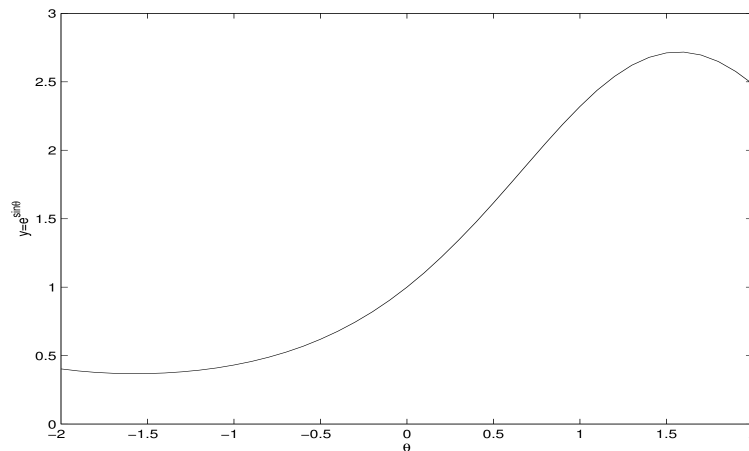

FIGURE 4.3 – Axes légendés avec des formules mathématiques.

On peut éviter les commandes en ligne en utilisant la barre de menu de la fenêtre de la figure. Il suffit de cliquer (avec le bouton de gauche de la souris) sur la flèche `edit plot` dans la barre d'outils, de la fenêtre de la figure, pour se mettre en mode sélection. Un clic-souris (bouton de droite) sur les axes permet de les sélectionner. Un autre clic-souris (bouton gauche) ouvre un menu. Il suffit de choisir `properties` dans ce menu pour pouvoir régler les axes, légender les axes, donner un titre à la figure, etc.

4.2.2 Les commentaires

Il est possible de donner un titre à la figure, de légender les courbes et d'insérer des commentaires directement dans la figure. Pour le titre, la commande

```
title('Titre de la figure')
```

permet de donner un titre à la figure.

Lorsqu'il y a plusieurs courbes sur la même figure, il faut une légende spécifique pour les distinguer. La fonction `legend` permet de légender chaque courbe. La syntaxe est

```
 legend('Courbe 1','Courbe 2',...)
```

L'ordre des légendes est celui des tracés.

Voici un exemple complet de courbes multiples légendées. Le résultat du bout de code suivant est la figure 4.4. Les courbes représentent les polynômes de Tchebychev

$$T_n(x) = \cos(n \arccos x), \quad x \in [-1,\ 1], \quad n = 0, 1, 2, 3.$$

```
>> x=-1:.01:1;
>> y0=ones(size(x)); y1=cos(acos(x));
>> y2=cos(2*acos(x)); y3=cos(3*acos(x));
>> plot(x,y0,':',x,y1,'-.',x,y2,'--',x,y3,'-')
>> xlabel('x')
>> ylabel('y=T_n(x)')
>> title('Polynomes de Tchebytchev')
>> legend('n=0','n=1','n=2','n=3')
```

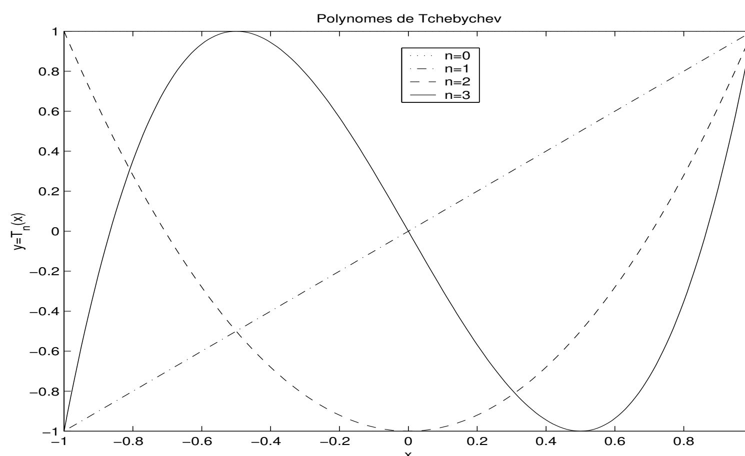

FIGURE 4.4 – Polynômes de Tchebychev pour $n = 0, 1, 2, 3$.

Pour insérer du texte dans une figure, il y a deux commandes `text` et `gtext`. En tapant

```
text(coordx,coordy,'Texte')
```

on insère la chaîne `'Texte'` dans la figure, au point d'abscisse `coordx` et d'ordonnée `coordy`. Il faut donc faire attention à la longueur du texte à insérer. La syntaxe pour `gtext` est

```
gtext('Texte')
```

Après le retour-charriot, une mire apparaît (avec la souris) dans la figure. Il suffit alors de déplacer la souris jusqu'à l'endroit voulu, un clic-souris permet d'insérer le texte en argument.

4.2.3 Décomposition de la fenêtre en sous-fenêtres

Il est possible de décomposer une fenêtre en plusieurs sous-fenêtres et d'afficher une figure différente dans chacune de ces sous-fenêtres grâce à la commande `subplot`. La syntaxe est

```
subplot(m,n,p)
```

où

- `m` est le nombre de sous-fenêtres verticales;
- `n` est le nombre de sous-fenêtres horizontales;
- `p` est la position de la sous-fenêtre de la figure à tracer.

Les sous-fenêtres sont numérotées de gauche à droite, de haut en bas. Voici un exemple avec les polynômes de Tchebychev tracés dans des sous-fenêtres séparées. Le résultat est la figure 4.5

```
>> x=-1:.01:1;
>> y1=cos(acos(x));
>> subplot(2,2,1)        % 1ere sous-fenetre
>> plot(x,y1)
>> title('Polyn\^ome de Tchebychev T_1(x)=cos(acos(x))')
>> xlabel('x')
>> ylabel('y=T_1(x)')
>> subplot(2,2,2)        % 2eme sous-fenetre
>> y2=cos(2*acos(x));
>> plot(x,y2)
>> title('Polyn\^ome de Tchebychev T_2(x)=cos(2 acos(x))')
>> ylabel('y=T_2(x)')
>> xlabel('x')
>> subplot(2,2,3)        % 3eme sous-fenetre
>> y3=cos(3*acos(x));
>> plot(x,y3)
>> title('Polyn\^ome de Tchebychev T_3(x)=cos(3acos(x))')
>> xlabel('x')
>> ylabel('y=T_3(x)')
>> subplot(2,2,4)        % 4eme sous-fenetre
>> y4=cos(4*acos(x));
>> plot(x,y4)
>> title('Polyn\^ome de Tchebychev T_4(x)=cos(4acos(x))')
>> ylabel('y=T_4(x)')
>> xlabel('x')
```

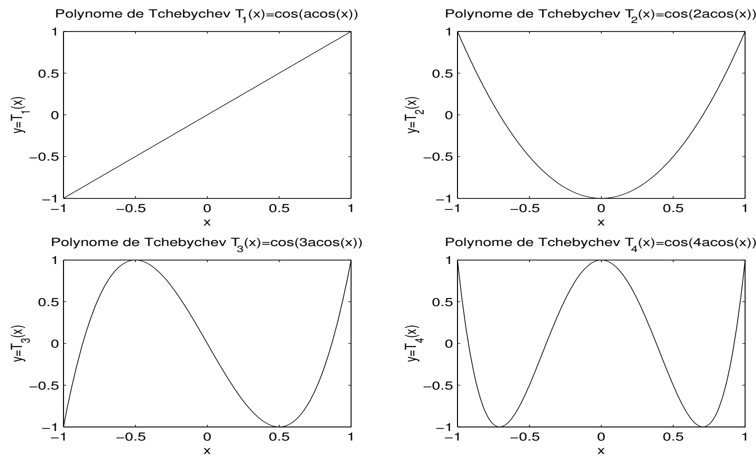

FIGURE 4.5 – Polynômes de Tchebychev pour $n = 1, 2, 3, 4$.

4.2.4 Sauvegarder une figure

La commande `print` permet de sauvegarder une figure sous divers formats. La syntaxe est

```
print -f<num> -d<format> <nomfich>
```

où

`<num>` est le numéro de la fenêtre graphique à sauvegarder. Par défaut (*i.e.* en l'absence de `-f<num>` dans la ligne de commande) c'est la fenêtre active qui est sauvegardée.

`<format>` spécifie le format de sauvegarde. Les formats les plus utilisés sont

- `ps` PostScript noir et blanc
- `ps2` PostScript noir et blanc, niveau 2
- `psc` PostScript couleur
- `psc2` PostScript couleur, niveau 2
- `eps` PostScript encapsulé noir et blanc
- `eps2` PostScript encapsulé noir et blanc, niveau 2
- `epsc` PostScript encapsulé couleur
- `epsc2` PostScript encapsulé couleur, niveau 2
- `tiff` TIFF
- `jpeg<nn>` JPEG de niveau de compression `nn`%, *i.e.* `-djpeg90` donne une image JPEG avec un niveau de compression de 90%. Par défaut le niveau de compression est de 75%.

`<nomfich>` le nom du fichier de sauvegarde, avec ou sans extension. L'extension par défaut est celle spécifiée dans le format.

A noter que pour les différents formats PostScript, l'extension est soit `.ps` soit `.eps`. Pour générer toutes les figures de ce manuel, nous avons utilisé `print -deps2` ou `print -depsc2`.

Pour éviter les commandes, on peut utiliser la barre de menus de la fenêtre de la figure. Il suffit de cliquer sur *file* puis sur *save* ou *save as* pour sauvegarder la figure au format voulu.

4.3 Visualisation 3D

Pour la visualisation 3D, il convient de distinguer les courbes 3D (plus faciles à tracer) des surfaces. Et pour les surfaces, il convient de distinguer celles définies de manière analytique (*i.e.* par une fonction) des surfaces définies par un ensemble de points de l'espace.

4.3.1 Courbes 3D

La commande `plot3` permet de tracer une courbe 3D. La syntaxe est

```
plot3(x,y,z)
```

où `x`, `y` et `z` peuvent être des matrices ou des vecteurs de *même taille*. Dans le cas ou les arguments `x`, `y` et `z` sont des vecteurs, la commande génère une ligne 3D qui passe par les points (x_i, y_i, z_i). Le bout de code suivant a permis de générer la courbe 3D de la figure 4.6.

```
>> t=0:pi/50:10*pi;
>> x=(1+sin(t)).*cos(t);y=(1+sin(t)).*sin(t); z=t;
>> plot3(x,y,z)
>> grid on
>> xlabel('x'),ylabel('y'),zlabel('z')
```

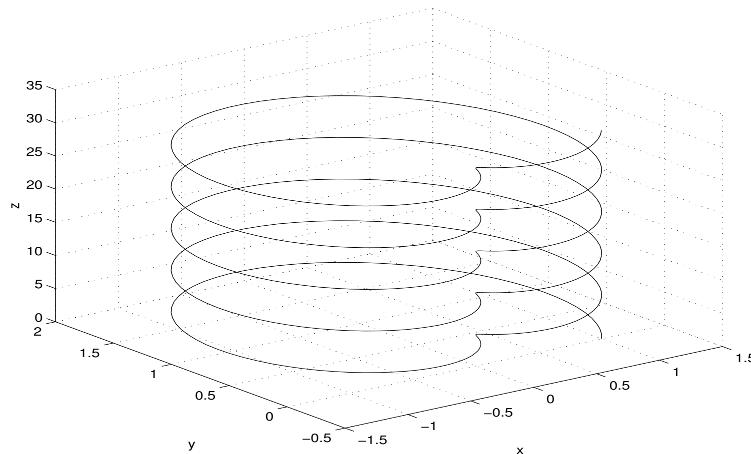

FIGURE 4.6 – Courbe 3D définie par $((1 + \sin t) \cos t, (1 + \sin t) \sin t, t)$, pour $t \in [0,\ 30\pi]$.

Si les arguments `x`, `y` et `z` sont des matrices de même dimension, `plot3` trace une courbe 3D pour chaque colonne des matrices. La surface de la figure 4.7 a été générée à l'aide du bout de code suivante.

```
>> [x,y]=meshgrid(-2:.1:2);
>> z=x.*exp(-x.^2-y.^2);
>> plot3(x,y,z)
```

Il suffit alors de tracer des courbes 3D suivant les lignes de `x`, `y` et `z` pour avoir la surface de la figure 4.8. Pour cela, on rajoute les lignes de codes.

```
>> hold on
>> plot3(x',y',z')
```

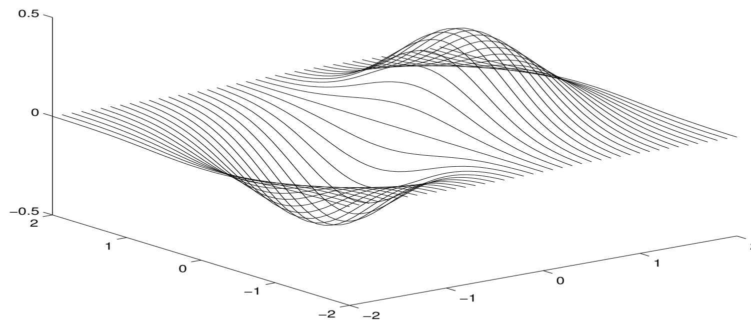

FIGURE 4.7 – Commande `plot3(x,y,z)` avec x, y et z des matrices avec la fonction $z = x\mathrm{e}^{-x^2-y^2}$.

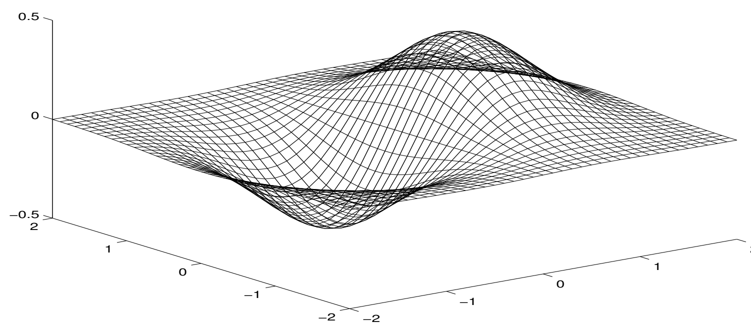

FIGURE 4.8 – Surface tracée avec `plot3` sur les colonnes, puis sur les lignes avec la fonction $z = x\mathrm{e}^{-x^2-y^2}$.

4.3.2 Surface analytique

On suppose ici que la surface est définie sous la forme

$$z = f(x, y).$$

On commence d'abord par générer une grille avec `meshgrid` dont la syntaxe est

```
[x,y] = meshgrid(x_min:hx:x_max,y_min:hy:y_max)
```

où `hx` est la raison de la subdivision sur l'axe des x et `hy` celle sur l'axe des y. Les arguments résultats `x` et `y` sont des matrices de même taille. Les lignes de `x` sont des copies du vecteur `x_min:hx:x_max`

et les colonnes de y sont des copies du vecteur x_min:hx:x_max. Si les subdivisions en x et y sont identiques, on utilise la forme abrégée

```
[x,y] = meshgrid(x_min:h:x_max)
```

Ensuite, il faut évaluer la fonction f aux noeuds de la grille pour avoir la troisième matrice z. Finalement on trace la surface avec mesh(x,y,z) (surface fil de fer) ou avec surf(x,y,z) (surface à facettes).

Soit à tracer la surface définie par

$$f(x,y) = x\mathrm{e}^{-x^2-y^2} \tag{4.1}$$

dans $[-2,\ 2]^2$. Les commandes suivantes permettent d'avoir la surface fil de fer de la figure 4.9

```
>> [x,y]=meshgrid(-2:.1:2);
>> z=x.*exp(-x.^2-y.^2);
>> mesh(x,y,z)
```

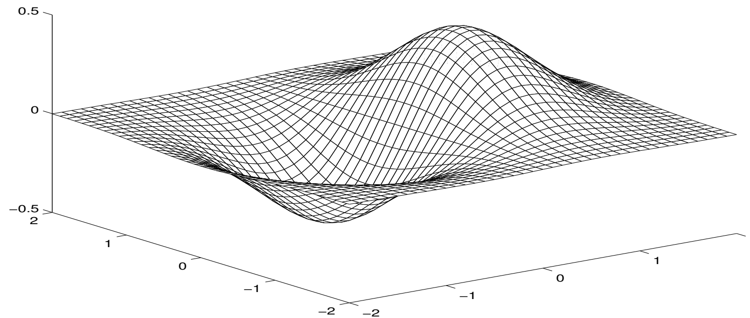

FIGURE 4.9 – Tracé fil de fer de la surface définie par (4.1)

Pour pouvoir apprécier une surface, il faut pouvoir la voir sous tous les angles! Avec MATLAB cela se fait soit avec la souris soit avec la commande view dont la syntaxe est

```
view(az,el)
```

où les arguments (en degrés) sont

az azimut, *i.e.* l'angle de rotation dans le plan horizontal

el élévation vertical, *i.e.* l'angle avec le plan horizontal

La figure 4.10 représente la surface définie par (4.1) avec view(30,60).

Les valeurs par défaut sont az=-37.5 et el=30. L'angle de vue par défaut s'obtient en tapant simplement view(3). A noter que view(2) donne l'angle de vision par défaut en 2D, à savoir az=0 et el=90. On regarde alors la surface à la verticale.

On peut aussi opter pour une visualisation plus interactive en utilisant la barre d'outils de la fenêtre de la figure. Il suffit de cliquer (bouton de gauche de la souris) sur *rotate 3D* pour mettre la figure en mode rotation. Un clic-souris (bouton de droite) continu combiné aux mouvements de la souris permet de tourner l'image dans l'espace.

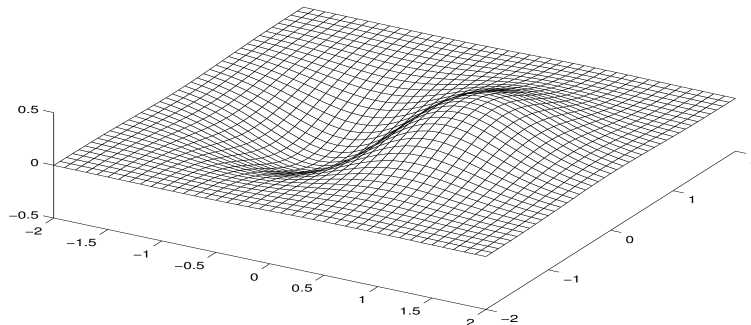

FIGURE 4.10 – Tracé fil de fer de la surface définie par (4.1), avec `view(30,60)`

4.3.3 Surface définie par un ensemble de points

MATLAB ne sait tracer que les surfaces définies par une grille de points. Mais en pratique, les surfaces à tracer sont souvent sous la forme d'un ensemble de points `x`, `y`, `z`. La visualisation de la surface, avec `mesh` nécessite, dans ce cas, un traitement préalable. Le but de ce traitement est de recréer une grille de points sur le plan Oxy par interpolation.

D'abord on récupère les coordonnées min et max pour reconstruire le domaine

```
>> xmin=min(x); xmax=max(x);
>> ymin=min(y); ymax=max(y);
```

Ensuite, à l'aide de `linspace`, on génère une subdivision uniforme en x et y.

```
>> xu=linspace(xmin,xmax,m);
>> yu=linspace(ymin,ymax,m);
```

où `m` est la taille du vecteur à générer. Il suffit maintenant de générer la grille avec `meshgrid` et de calculer (par interpolation) la valeur de la fonction aux noeuds de la grille.

```
>> [X,Y]=meshgrid(xu,yu);           % génération de la grille X,Y
>> Z=griddata(x,y,z,X,Y,'cubic');  % calcul de Z par interpolation
>> mesh(X,Y,Z)                      % visualisation
```

La fonction `griddata` calcule `Z` aux noeuds de la grille `[X,Y]` par interpolation *cubique* à partir de `[x,y,z]`. D'autres types d'interpolation sont disponibles. Par défaut, l'interpolation est linéaire.

Comme exemple, voici la surface définie par (4.1) tracée à partir d'un ensemble de points aléatoires. Le résultat, Fig. 4.11, n'est pas si mauvais.

```
>> x=rand(100,1)*4-2;
>> y=rand(100,1)*4-2;
>> z=x.*exp(-x.*x-y.*y);
>> xu=linspace(min(x),max(x),50);
>> xu=linspace(min(y),max(y),50);
```

```
>> [X,Y]=meshgrid(xu,yu);
>> Z=griddata(x,y,z,X,Y,'cubic');
>> C=zeros(size(Z));
>> mesh(X,Y,Z,C)
>> grid off;  axis tight; hold on
>> plot3(x,y,z,'*')
```

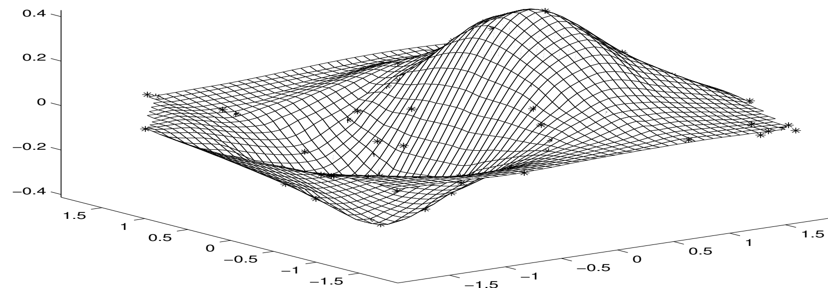

FIGURE 4.11 – Tracé fil de fer d'une surface obtenue par un ensemble de points. Les noeuds d'interpolation sont indiqués par des $\star$

Une autre façon de tracer une surface définie par un ensemble de points est de passer par une triangulation. On utilise la fonction MATLAB `delaunay` qui effectue la triangulation de Delaunay (2D) d'un ensemble de points qu'on visualise avec la fonction `triplot`.

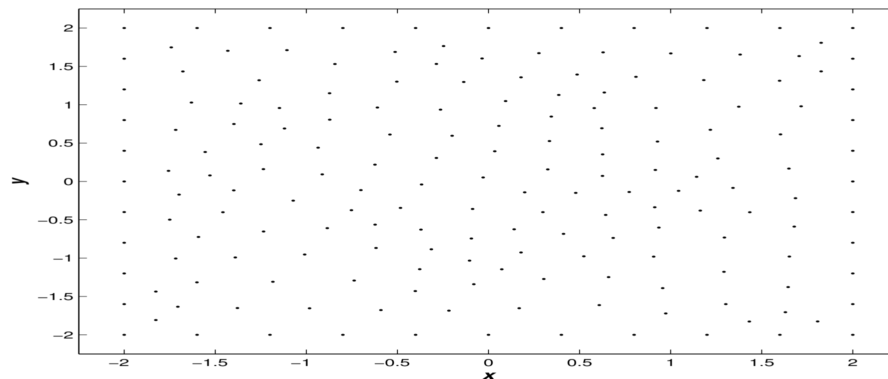

FIGURE 4.12 – Nuage de points

Le nuage de points de la figure 4.12 est triangulé avec la fonction `delaunay` comme suit.

```
>> t=delaunay(x,y);
>> triplot(t,x,y)
```

La triangulation obtenue est visualisée dans la figure 4.13. La fonction `triplot` permet de visualiser une triangulation 2D. Après la triangulation, il suffit de tracer la surface définie aux sommets de la triangulation. On obtient alors une surface définie par des facettes.

```
>> z=x.*exp(-x.^2-y.^2);
>> trisurf(t,x,y,z,'facecolor','interp')
```

Le résultat est la figure 4.14. La couleur est interpolée sur la surface grâce aux deux derniers arguments. Sinon la couleur est constante sur chaque facette. On peut aussi visualiser la surface d'en haut (avec `view(2)`) en activant la barre de couleur pour repérer les pics.

```
>> colormap(1-gray)                       % tracé en niveaux de gris inversé
>> trisurf(t,x,y,z,'facecolor','interp')
>> view(2)                                % vue d'en haut
>> colorbar('South')                      % barre de couleurs horizontale
```

Le résultat est la figure 4.15.

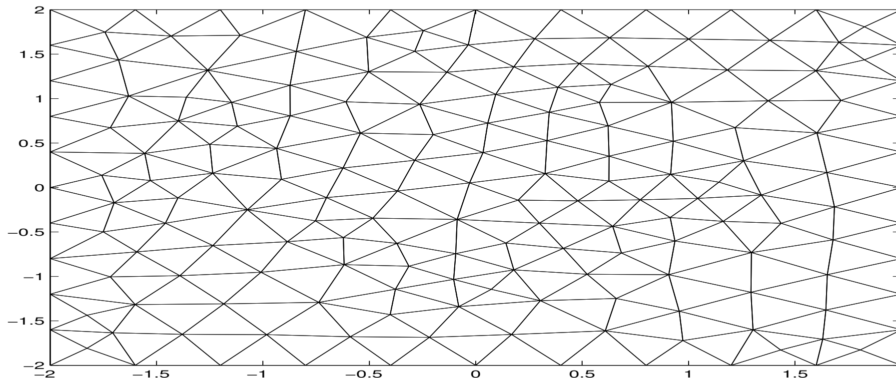

FIGURE 4.13 – Triangulation de Delaunay d'un ensemble de points

4.3.4 Courbes de niveau

Parfois, il est plus utile de tracer les courbes (ou lignes) de niveau que la surface elle-même. C'est particulièrement le cas lorsqu'on veut visualiser une distribution de pression (isobare), de température (isotherme), de contraintes, etc. Le tracé des courbes de niveaux se fait par la fonction `contour` dont la syntaxe est

```
contour(x,y,z [,n])
```

où `x`,`y`,`z` représente le maillage de la surface et `n` le nombre de courbes de niveau à tracer. Par défaut, *i.e.* en l'absence de `n`, le nombre de courbe est calculé en fonction des valeurs minimale et maximale prises par la fonction.

Pour tracer les courbes de niveau d'une fonction $f(x, y)$ on procède de la même manière que pour le tracé de la surface. Par exemple, pour tracer les courbes de niveau de la fonction

$$f(x, y) = xy\mathrm{e}^{-x^2-y^2}, \quad (x, y) \in [-2,\ 2] \times [-2,\ 2] \tag{4.2}$$

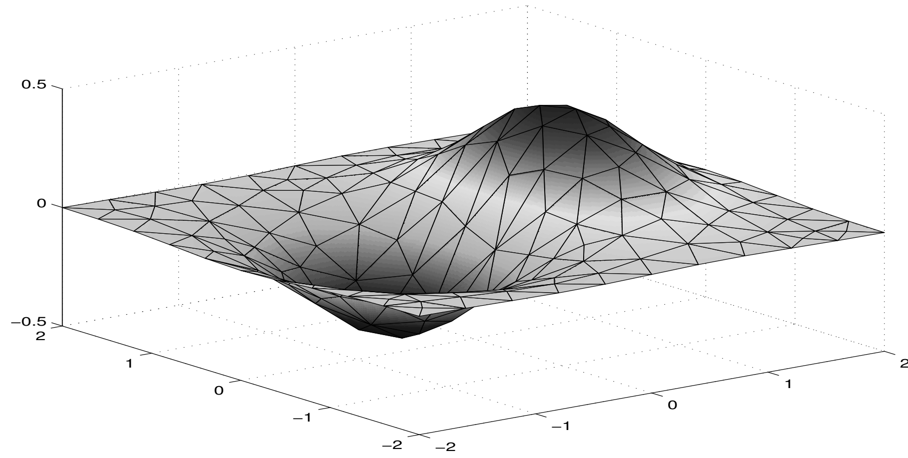

FIGURE 4.14 – Tracé par facettes d'une surface définie sur la triangulation 4.13

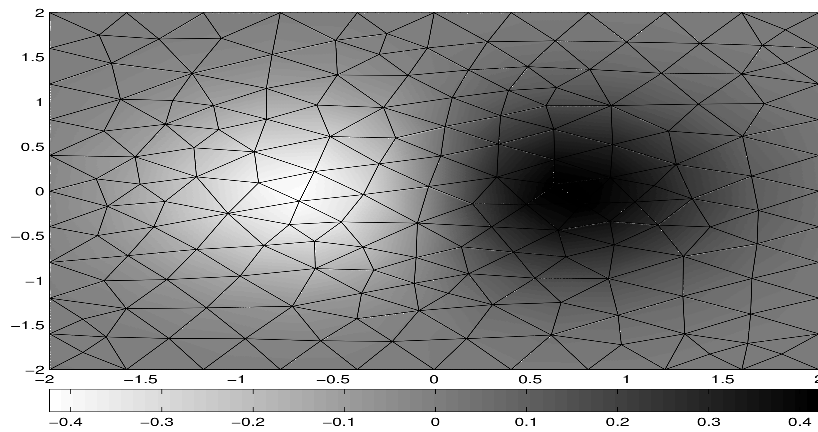

FIGURE 4.15 – Tracé par facettes d'une surface définie sur la triangulation 4.13

on procède comme suit.

```
>> [x,y]=meshgrid(-2:.05:2);
>> z=x.*y.*exp(-x.^2-y.^2);
>> contour(x,y,z,15)          % on trace 15 courbes de niveau
```

On obtient alors les courbes de niveau de la figure 4.16

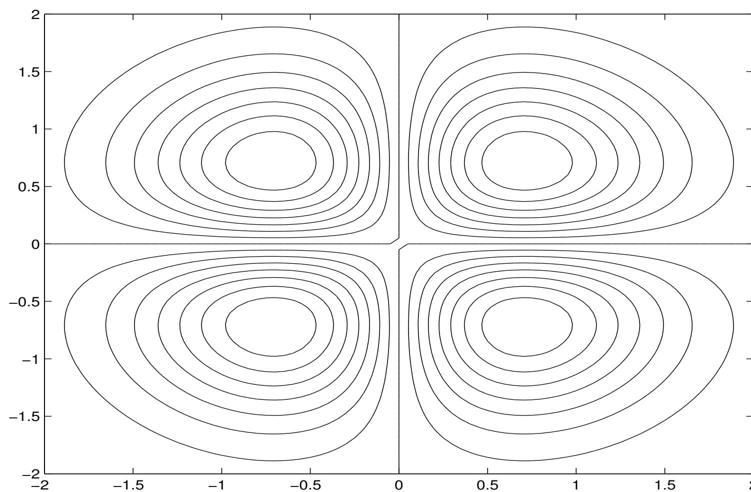

FIGURE 4.16 – Courbes de niveau de la surface définie par (4.2)

Pour afficher la valeur de la courbe de niveau sur la figure, on utilise la fonction `clabel` (*contour label*). Pour cela il faut d'abord récupérer la valeur des courbes de niveau. On procède comme suit (avec la fonction (4.2) déjà tabulée)

```
>> [c,h]=contour(x,y,z);
>> clabel(c,h)
```

On obtient les courbes de niveau de la figure 4.17

A noter que seules les courbes suffisamment «longues» reçoivent une valeur. Celles qui sont trop «petites» sont ignorées.

La variante

```
clabel(c,h,'manual')
```

permet d'afficher la valeur de certaines courbes de niveau qu'on sélectionne grâce à la souris. Par exemple, avec la fonction (4.2), si on exécute

```
>> [c,h]=contour(x,y,z,20);
>> clabel(c,h,'manual')
```

on obtient, après sélection avec la souris, la figure 4.18.

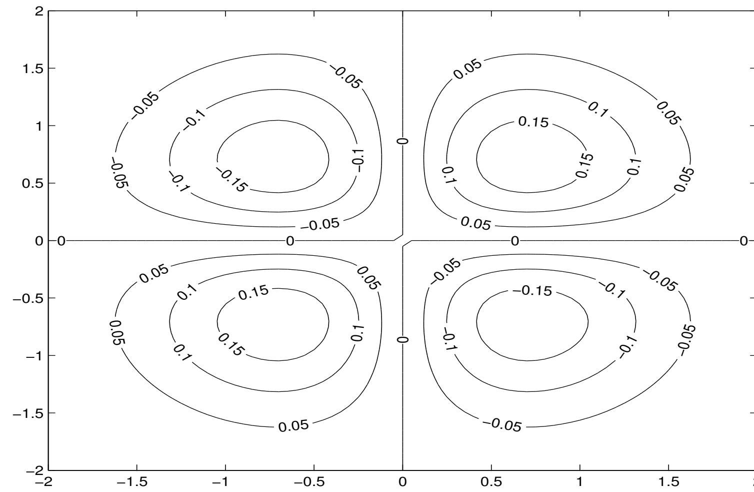

FIGURE 4.17 – Courbes de niveau avec valeurs de la surface définie par (4.2)

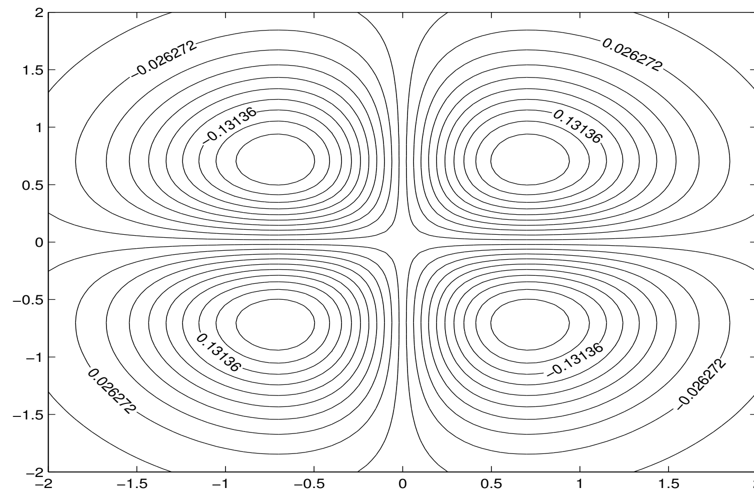

FIGURE 4.18 – Courbes de niveau avec valeurs sélectionnées de la surface définie par (4.2)

4.4 Exercices

Exercice 4.1 Tracer la courbe de la fonction f définie par

$$f(x) = Kx^3, \quad x \in [-10,\ 10]$$

pour n valeurs quelconques de K dans $[-5,\ 5]$, $n \geq 5$.

Exercice 4.2 Pour $x, y \in [0, 2\pi]$, représenter les courbes suivantes

$$\begin{aligned} z_1(x,y) &= \sin(x) + \cos(y), \\ z_2(x,y) &= \sin(x)\cos(y), \\ z_3(x,y) &= \sin^2(x) - \cos^2(y), \end{aligned}$$

comme sur la figure 4.19.

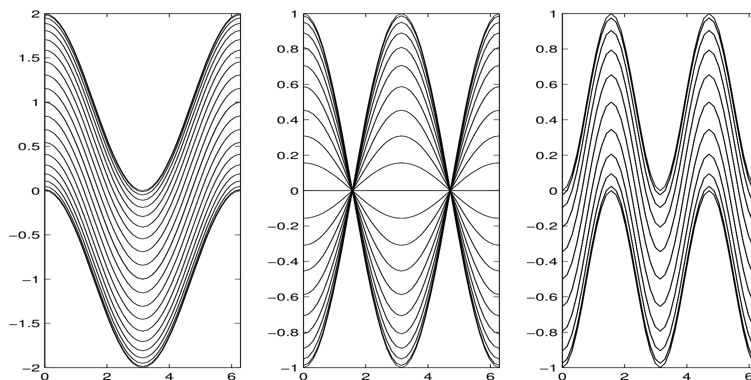

FIGURE 4.19 – Trajectoires z_1, z_2 et z_3 dans le plan

Exercice 4.3 (Courbes de niveau sur une surface) Les instructions suivantes ont permis d'avoir la surface de la figure 4.20, sans les courbes de niveau en blanc.

```
>> [X,Y] = meshgrid([-2:.25:2]);
>> Z = X.*exp(-X.^2-Y.^2);
>> surf(X,Y,Z,'facecolor','interp')
>> colormap bone
```

En utilisant la fonction `contour3` compléter la figure avec 30 courbes de niveau en blanc.

Exercice 4.4 Représenter, dans $[-1,\ 1] \times [-1,\ 1]$, la fonction

$$f(x) = \frac{\sin\sqrt{x_1^2 + x_2^2}}{\sqrt{x_1^2 + x_2^2}}$$

en utilisant une grille relativement grossière et sans précaution particulière sur de dénominateur de la fraction. Que constatez vous?

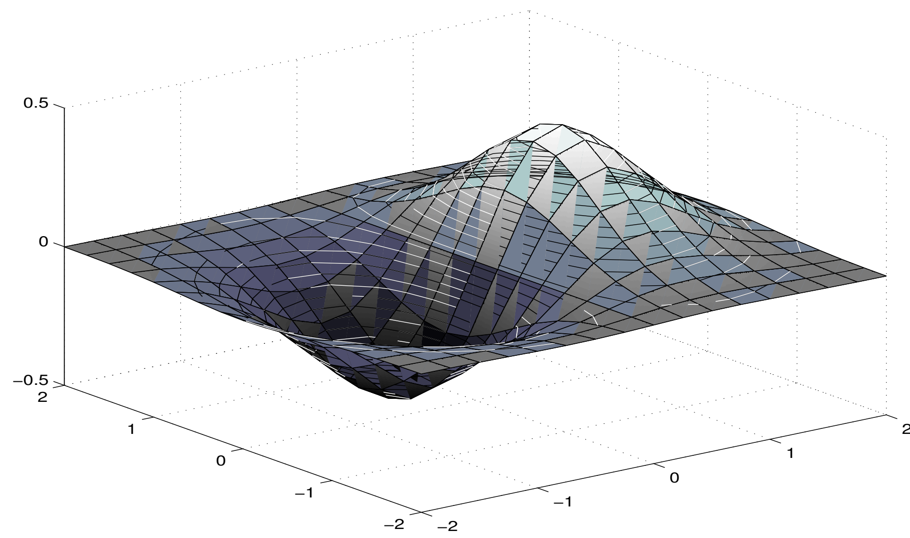

FIGURE 4.20 – Lignes de niveaux sur une surface

CHAPITRE 5

MATRICES CREUSES ET MÉTHODES ITÉRATIVES

Une matrice creuse ou peu dense (*sparse* en anglais) est une matrice ayant une faible densité d'éléments non nuls. C'est le cas des matrices générées par les méthodes des éléments ou des différences finis pour la résolution des problèmes d'équations aux dérivées partielles. Pour ces matrices, MATLAB propose de ne stocker que les éléments non nuls et leurs indices. D'où un gain d'espace mémoire et de temps de calcul en éliminant les opérations sur les éléments nuls. Ce mode stockage est particulièrement adapté aux méthodes itératives de résolution de systèmes linéaires.

5.1 Mode de stockage

La méthode de stockage d'une matrice creuse utilisée par MATLAB est la méthode CSR (*Compressed Sparse Rows*), *i.e.* une compression de lignes de la matrice par élimination d'éléments nuls. Les lignes compressées sont ensuite mises bout à bout pour former le vecteur de stockage. Pour une matrice de taille $m \times n$, on a besoin de 3 vecteurs `S`, `is` et `js` pour organiser le stockage.

`S` Le vecteur de stockage qui contient les lignes compressées de la matrice mises bout à bout. La longueur de `S` est le nombre d'éléments non nuls (ou susceptibles de l'être) de la matrice d'origine. A noter que c'est ce vecteur `S` qui est «visible» par l'utilisateur MATLAB.

`is` Vecteur (d'entiers) de longueurs $m + 1$ qui contient les indices de début de ligne dans le vecteur de stockage `S`. Ce vecteur n'est pas visible par l'utilisateur MATLAB.

`js` Vecteur (d'entiers) contenant les indices de colonnes d'éléments non nuls. Ce vecteur de la même taille que `S` est aussi invisible par l'utilisateur.

Une matrice pleine `A` peut-être convertie en un matrice creuse `S` et *vice versa* grâce aux fonctions `sparse` et `full`

```
S=sparse(A)     % conversion : A matrice pleine  -> S matrice creuse
A=full(S)       % conversion : S matrice creuse -> A matrice pleine
```

5.2 Création

On peut créer une matrice creuse, grâce à la fonction `sparse`, en listant les éléments non nuls et leurs indices

```
S=sparse(ii,jj,s,m,n)
```

avec

`ii,jj` vecteurs de même longueur contenant les indices de lignes et colonnes d'éléments non nuls;

`s` vecteur, de même longueur que `ii` et `jj`, contenant les valeurs des éléments non nuls de la matrice;

`m,n` nombre de lignes et de colonnes de la matrice d'origine.

Voici un exemple simple de création d'une matrice creuse 25×20.

```
>> ii=[2 7 10 11 12 15];
>> jj=[1 6 10 12 12 17];
>> s=rand(1,6)*10;
>> S=sparse(ii,jj,s,25,20)
S =
   (2,1)       9.50129285147175
   (7,6)       2.31138513574288
  (10,10)      6.06842583541787
  (11,12)      4.85982468709300
  (12,12)      8.91298966148902
  (15,17)      7.62096833027395
```

Un sixième argument à la fonction `sparse` permet de réserver plus de place que d'éléments non nuls par exemple pour des affectations ultérieures. L'instruction complète de création de matrice creuse est donc

```
S=sparse(ii,jj,s,m,n,nzmax)
```

où `nzmax` est le nombre maximal d'éléments non nuls. La fonction `nnz` permet de connaître le nombre d'éléments non nuls d'une matrice creuse, qui peut être différent de la longueur du vecteur de stockage.

Remarque 5.1 *La fonction* `sparse` *est aussi utilisée pour l'assemblage de matrices dans la méthode des éléments finis.*

Certaines fonctions MATLAB disposent de variantes pour les matrices creuses, qu'on peut utiliser pour la génération de matrices creuses

`spdiags` variante matrice creuse de la fonction `diag`;

`speye` variante matrice creuse de la fonction `eye`;

`sparse(m,n)` équivalent matrice creuse de la fonction `zeros(m,n);`

`sprand(m,n,d), sprandn(m,n,d)` matrices creuses de densité `d` générées suivant les lois uniforme et normale;

`sprandsym(n,d)` matrice creuse aléatoire, carrée symétrique d'ordre `n` et de densité `d`.

La fonction `spfun` permet d'évaluer une fonction sur les seuls éléments non nuls d'une matrice creuse.

```
>> tic, C=cos(A); toc
Elapsed time is 2.921456 seconds.

>> c=@cos;
>> tic, C=spfun(c,A); toc
Elapsed time is 0.063000 seconds.
```

La combinaison des fonctions `tic` (démarre l'horloge) et `toc` (arrête l'horloge) permet d'évaluer le temps de calcul. Le symbole `@` définit une variable de type `function handle`, *cf.* § 6.4.

Pour une matrice creuse `S`, la fonction `find` permet de récupérer les indices et les valeurs des éléments non nuls

```
[ii,jj,s]=find(S)
```

La fonction `spy` permet de visualiser une matrice creuse. La Figure 5.1 montre la répartition des éléments non nuls de la matrice creuse (prédéfinie) `west0479`

```
>> load west0479
>> spy(west0479)
>> densite=nnz(west0479)/479^2
densite =
    0.0082
```

La faible densité de la matrice `west0479` justifie son stockage en matrice creuse.

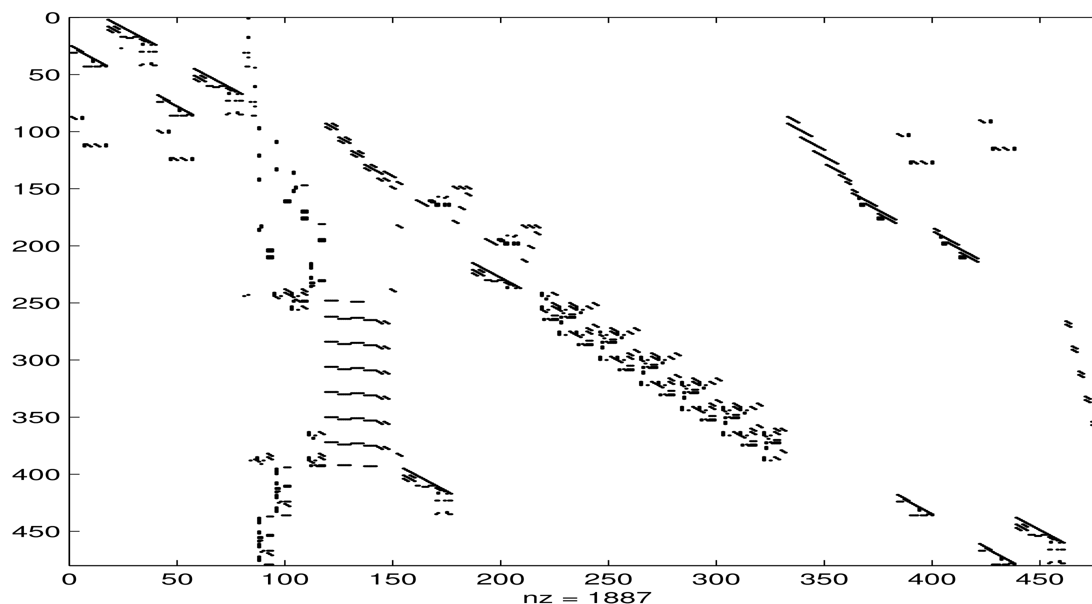

FIGURE 5.1 – Visualisation de la matrice creuse `west0479`

Une fois créée, la matrice est statique, *i.e.* il n'y a pas de mise à jour de la taille même si des éléments deviennent nuls.

5.3 Opérations sur les matrices creuses

En principe pour les opérations binaires, si les deux opérandes sont des matrices creuses ou pleines, le résultat est une matrice creuse ou pleine. Dans le cas où l'un des opérandes est une matrice creuse et l'autre une matrice pleine, le résultat est une matrice dense à moins que l'opérateur préserve la densité. Soit F une matrice pleine et S une matrice creuse, alors

`S+F` matrice pleine;

`S*F` matrice pleine;

`F\S` matrice pleine;

`S.*F` matrice creuse;

`[S F]` matrice creuse.

5.4 Factorisation et méthodes directes

Une matrice creuse peut être factorisée moyennant un remplissage (*fill-in* en Anglais). Nous n'étudions dans cette section que les factorisations LU et de Cholesky.

5.4.1 Factorisation LU

Si `S` est une matrice creuse, sa factorisation LU est obtenue à l'aide de la fonction `lu` (§3.4)

```
[L,U]=lu(S)
```

ou, dans le cas où on veut isoler les permutations

```
[L,U,P]=lu(S)
```

Les facteurs `L` et `U` sont les mêmes que si `S` était une matrice pleine, sauf qu'ici ce sont des matrices creuses. Un deuxième argument permet de mieux contrôler le pivotage

```
[L,U]=lu(S,ptol)
```

où $0 \leq \texttt{ptol} \leq 1$. La recherche du pivot ne s'effectue que si l'élément diagonal est (en valeur absolue) `ptol` fois plus petit que les éléments de la sous-colonne, *i.e.*

$$|S_{ii}| < \texttt{ptol}|S_{ij}|, \quad i = j+1, \ldots, n.$$

La valeur par défaut de `ptol` est 1.

Voici, comme exemple la factorisation de la matrice creuse `west0479` de MATLAB. On remarque la variation du nombre d'éléments non nuls en fonction du facteur de contrôle de pivotage.

```
>> load west0479
>> dwest0479=nnz(west0479)/479^2  % densite de west0479
dwest0479 =
    0.0082

>> [L,U]=lu(west0479);            % factorisation LU
>> dL=nnz(L)/479^2                % densité de la matrice L
```

```
dL =
    0.0495

>> dU=nnz(U)/479^2                  % densité de la matrice U
dU =
    0.0287

>> [L,U]=lu(west0479,.45);          % LU avec contrôle pivotage
>> dL=nnz(L)/479^2                  % densité de la matrice L
dL =
    0.0469

>> dU=nnz(U)/479^2                  % densité de la matrice U
dU =
    0.0323
```

La factorisation induisant un remplissage (*i.e.* des éléments nuls devenant non nuls), les matrices `L` et `U` comportent donc beaucoup plus d'éléments non nuls que la matrice d'origine `west0479` comme le montre la figure 5.2. La densité des matrices L et U donne une idée du coût de la factorisation.

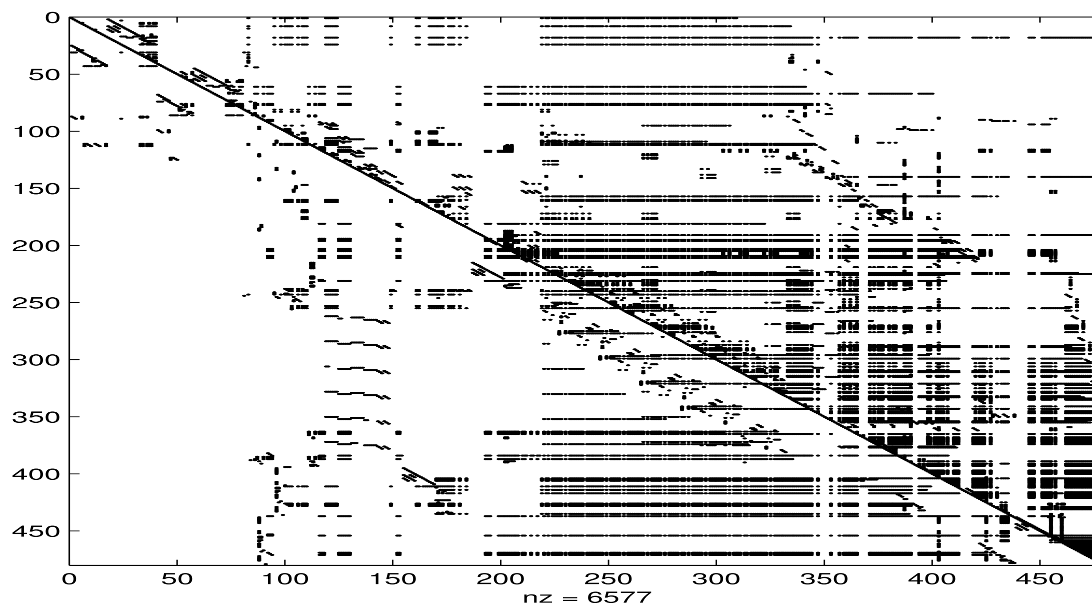

FIGURE 5.2 – Visualisation des matrices L et U de la factorisation LU de `west0479`

La résolution du système, avec la matrice creuse factorisée, est semblable à celle d'un système à matrice dense, *i.e.*

```
x=U\(L\b);
```

5.4.2 Factorisation de Cholesky

Si une matrice creuse `A` est symétrique, définie positive, on peut la mettre sous forme RR^T (`R'*R` pour MATLAB) grâce à la factorisation de Cholesky (*cf.* §3.4)

```
R=chol(A)
```

Le facteur `R` est une matrice creuse. La résolution du système $Ax = b$ est semblable à celle d'un système à matrice dense, *i.e.*

```
x=R\(R'\b)
```

La matrice du Laplacien est un exemple de matrice (creuse) définie positive. La fonction MATLAB `delsq` construit la matrice (creuse) du Laplacien 2D par différences finies à 5 points, voir par exemple [11, 13]. La fonction `numgrid` génère une grille de points dans un domaine bidimensionnel (voir l'aide en ligne pour le type de domaines). Dans l'exemple suivant, on génère la matrice creuse du Laplacien puis on la factorise. On remarque la différence (considérable) de densité entre la matrice d'origine `A` et son facteur `R`. Le remplissage, dû à la factorisation, est dans ce cas très important.

```
>> A=delsq(numgrid('S',100));
>> size(A)
ans =
        9604        9604

>> R=chol(A);
>> dA=nnz(A)/9604^2            % densité A
dA =
   5.1637e-04

>> dR=nnz(R)/9604^2            % densité du facteur R
dR =
    0.0102
```

5.5 Permutation des lignes et des colonnes

En permutant les lignes et/ou les colonnes d'une matrice, on peut obtenir des facteurs LU ou de Cholesky moins denses qu'avec la matrice d'origine. Dans le cas d'une matrice issue de la méthode des éléments finis (ou de différences finies), cela revient à renuméroter les sommets du maillage (ou de la grille).

Plusieurs fonctions permettent d'effectuer la permutation des lignes et/ou des colonnes d'une matrice (`amd`, `colperm`, `colamd`, `colmmd`, `symrcm`, `symamd`). Mais nous n'étudions dans cette section que deux méthodes, largement utilisées dans les cours d'éléments finis : la *méthode inverse de Cuthill-McKee* et la *méthode du degré minimum*.

La méthode de Cuthill-Mckee (voir, par exemple, [11, 15]) produit une matrice dont les éléments non nuls sont concentrés près de la diagonale en réduisant la largeur de bande. La matrice résultat est donc «proche» d'une matrice bande.

En théorie des graphes, le degré d'un sommet est le nombre de ses voisins. Pour la matrice d'adjacence, cela correspond aux nombres d'éléments (hors diagonaux) non nuls. La méthode du degré mi-

nimum a pour but de réduire (localement) le remplissage (voir, par exemple, [1, 15]). Ce qui a pour conséquence des facteurs de Cholesky moins denses que ceux obtenus sans permutation.

Les fonctions MATLAB correspondantes sont

- `p=symrcm(A)` permutation avec la méthode inverse de Cuthill-McKee, pour matrices symétriques ou non;
- `p=amd(A)` permutation avec la méthode du degré minimum approché. L'algorithme AMD (voir par exemple [4]) est appliqué à la matrice $C = A + A^T$. La matrice A doit être symétrique ou avoir les éléménts nons nuls repartis de manière symétrique.
- `p=symamd(A)` permutation avec la méthode du degré minimum *symétrique* approché pour les matrices symétriques définies positives.

Une fois le vecteur de permutation obtenu, il suffit de remplacer, dans les opérations impliquant la matrice, `A` par `A(p,p)`. L'exemple suivant reprend la matrice `west0479` de matlab.

```
>> load west0479
>> p1=symrcm(west0479);          % permutations  avec symrcm
>> [L1,U1]=lu(west0479(p1,p1));  % factorisation LU
>> dL1=nnz(L1)/479^2             % densité L
dL1 =
    0.0254

>> dU1=nnz(U1)/479^2              % densité U
dU1 =
    0.0348
```

La Figure 5.3 montre la repartition des éléments de la matrice `west0479` après permutation de lignes et de colonnes avec la fonction `symrcm`.

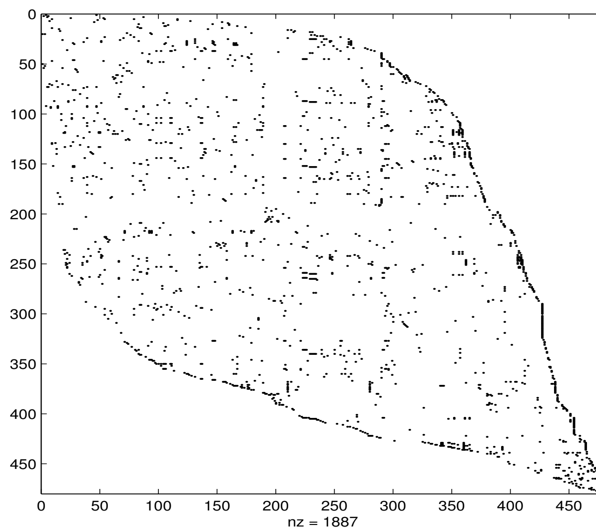

FIGURE 5.3 – Matrice `west0479` permutée avec `symrcm`

Pour la résolution, il faut tenir compte des permutations, *i.e.* permuter le second membre et les composantes du vecteur solution.

```
>> load west0479;
>> p=symrcm(west0479);              % permutation
>> [L,U]=lu(west0479(p,p));         % factorisation
>> b=ones(479,1);                   % second membre
>> x(p)=U\(L\b(p));                 % résolution
>> size(x)
ans =
     1   479

>> norm(west0479*x'-b)              % verification
ans =
   1.8061e-10
```

On remarque que le vecteur solution est maintenant un vecteur ligne. Pour éviter ce désagrément (qui peut provoquer des erreurs fatales à l'exécution) il faut initialiser le vecteur solution avant de le permuter. Il suffit donc de remplacer la ligne de résolution ci-dessus par

```
>> x=zeros(479,1); x(p)=U\(L\b(p));             % résolution
>> size(x)
ans =
     479   1
```

La matrice du Laplacien (avec condition aux bords de Dirichlet) vu au § 5.4.2 est symétrique définie positive. On peut donc la permuter avec les fonctions `amd` et `symamd`

```
>> A=delsq(numgrid('S',100)); n=size(A,1)
n =
   9604

>> R=chol(A); dR=nnz(R)/n^2  % densité de R
dR =
    0.0102

>> p=amd(A);                % permutation avec AMD
>> Rp=chol(A(p,p));
>> dRp=nnz(Rp)/n^2          % densité de R avec la permutation AMD
dRp =
    0.0021

>> q=symamd(A);             % permutation avec SYMAMD
>> Rq=chol(A(q,q));
>> dRq=nnz(Rq)/n^2          % densité de R avec la permutation SYMAMD
dRq =
    0.0020
```

Les Figures 5.4-5.5 montrent la répartition des éléments dans les matrices initiale et permutée. Pour la résolution, on procède comme pour la matrice `west0479`

```
>> b=ones(9604,1);
>> x=zeros(9604,1); x(p)=R\(R'\b(p));
>> norm(A*x-b)
ans =
   3.0860e-11
```

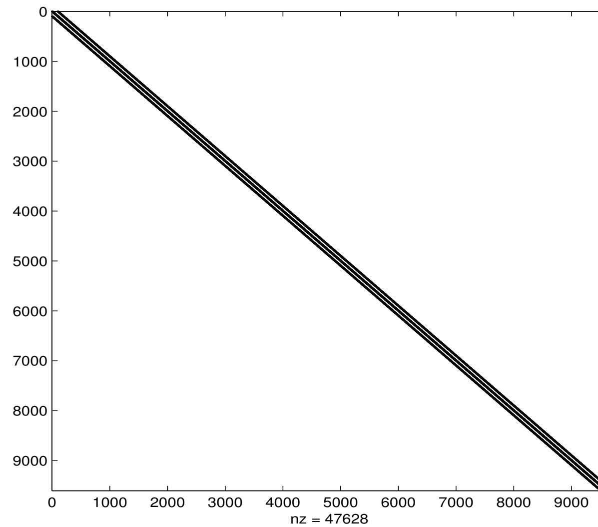

FIGURE 5.4 – Visualisation de la matrice du Laplacien générée par `delsq`

5.6 Factorisation incomplète et préconditionnement

La conditionnement d'une matrice carrée inversible A est le nombre

$$\sigma(A) = \| A \| \| A^{-1} \| \tag{5.1}$$

où $\| \cdot \|$ est une norme matricielle. On dit que A est bien conditionnée si $\sigma(A)$ est «proche» de 1.

Le préconditionnement d'un système linéaire

$$Ax = b \tag{5.2}$$

consiste à le remplacer par le système

$$M^{-1}Ax = M^{-1}b, \tag{5.3}$$

où M est choisie de telle sorte que $\kappa(M^{-1}A) << \kappa(A)$. On déduit immédiatement, d'après (5.1), que M^{-1} doit être une approximation (peu coûteuse) de A^{-1}, si on veut un meilleur conditionnement de (5.3).

Pour un x donné, le reste ou résidu r du système (5.2) est

$$r = Ax - b$$

tandis que celui du système préconditionné (5.3) est donné par

$$z = M^{-1}r. \tag{5.4}$$

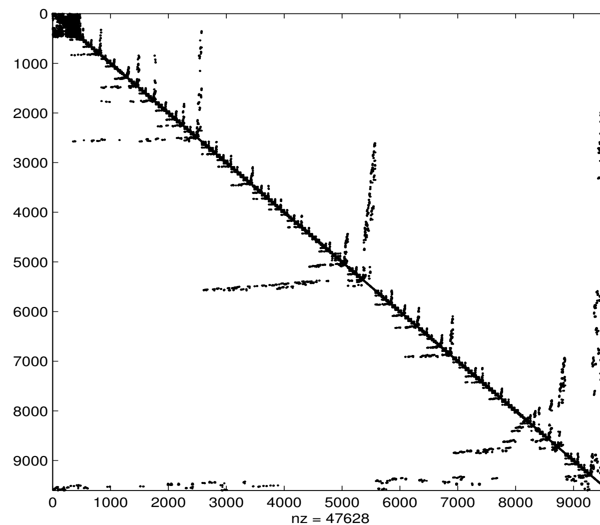

FIGURE 5.5 – Visualisation de la matrice du Laplacien, générée par `delsq` et permutée avec `symamd`

Comme M^{-1} est trop coûteux à calculer, on remplace (5.4) par le système

$$Mz = r \tag{5.5}$$

C'est un système du type (5.5) qui est résolu dans les méthodes itératives préconditionnées pour calculer le résidu. Comme signalé plus haut, la matrice M doit être une approximation de A pour que le système (5.3) soit bien conditionné. D'où l'idée de calculer une approximation de la factorisation de A: c'est la factorisation incomplète.

Les fonctions MATLAB `ilu` et `ichol` fournissent une factorisation incomplète (LU ou de Cholesky). Pour `ichol`, la matrice à factoriser doit être évidemment symétrique définie positive.

5.6.1 Factorisation LU incomplète

Pour la factorisation LU incomplète, la fonction est `ilu` [1] dont la syntaxe est

```
[L,U]=ilu(A,setup)
```

où `setup` est une structure à cinq champs (`type`, `droptol`, `milu`, `udiag` et `thresh`). En général les valeurs par défaut sont suffisantes sauf pour `droptol` car c'est cette valeur qui détermine le remplissage. En effet, un élément `A(i,j)` n'est conservé, lors de la factorisation, que si

```
abs(A(i,j))>setup.droptol*norm(A(:,j))
```

La factorisation s'effectue colonne après colonne pour limiter l'occupation mémoire. Évidemment avec `setup.droptol=0`, on obtient la factorisation exacte de A.

1. `ilu` remplace `luinc`

```
>> load west0479;                % matrice west0479
>> A=10^3*speye(479)+west0479;   % renforcement de la diagonale
>> n=size(A,1)
n =
   479

>> [L,U]=lu(A);                  % factorisation LU exacte de A
>> dL=nnz(L)/n^2                 % densité du facteur exact L
dL =
    0.0622

>> dU=nnz(U)/n^2                 % densité du facteur exact U
dU =
    0.0693

>> setup.droptol=0.001;          % facteur de remplissage
>> [Li,Ui]=ilu(A,setup);         % factorisation LU incomplète
>> dLi=nnz(Li)/n^2               % densité du facteur incomplet Li
dLi =
    0.0079

>> dUi=nnz(Ui)/n^2               % densité du facteur incomplet Ui
dUi =
    0.0044
```

On remarque les facteurs incomplets sont dix fois moins denses que les facteurs L et U exacts. Le choix du paramètre de remplissage `setup.droptol` n'est pas toujours évident. Plusieurs essais sont parfois nécessaires.

Les facteurs incomplets sont utilisés comme préconditionneurs dans les méthodes itératives de résolution de système linéaire comme la méthode du résidu minimum généralisé (fonction MATLAB `gmres` étudiée au §5.7.2) ou la méthode du bi-gradient conjugué (fonction MATLAB `bicg`).

5.6.2 Factorisation de Cholesky incomplète

Pour la factorisation de Cholesky incomplète, la fonction est `ichol` [2] dont la syntaxe est

```
R=ichol(A,opts)
```

où `opts` est une structure cinq champs (`type`, `droptol`, `michol`, `diagcomp`, `shape`). Les valeurs par défaut des trois derniers champs sont suffisantes pour les calculs de ce livre. Le champ `type` indique le type de factorisation:

- `'nofill'` effectue la factorisation de Cholesky sans remplissage (*i.e.* les éléments nuls restent nuls);
- `'ict'` (pour *incomplete Cholesky with threshold dropping*) effectue la factorisation de Cholesky avec un seuil de remplissage, comme pour `ilu`.

La valeur par défaut de ce champ est `nofill`. Lorsque `opts.type='ict'`, un élément `(`A(i,j)) n'est conservé que s'il vérifie

2. `ichol` remplace `cholinc`

```
abs(A(i,j))>opts.droptol*norm(A(j:end,:))
```

La valeur de `droptol` est ignorée si `opts.type='nofill'`. La variante

```
R=ichol(A)
```

effectue la factorisation sans remplissage.

```
>> A=delsq(numgrid('S',100));
>> n=size(A,1)
n =
    9604

>> R=chol(A);                    % factorisation de Cholesky exacte
>> dR=nnz(R)/n/n                 % densité du facteur exact R
dR =
    0.0102

>> opts.type='nofill';
>> R0=ichol(A,opts);              % factorisation de Cholesky sans remplissage
>> dR0=nnz(R0)/n/n
dR0 =
   3.1024e-04

>> opts.type='ict'; opts.droptol=0.001;
>> R1=ichol(A,opts);              % factorisation de Cholesky incomplète
>> dR1=nnz(R1)/n/n
dR1 =
    0.0013
```

Le choix du paramètre `opts.droptol` n'est pas toujours évident. Plusieurs essais sont parfois nécessaires pour obtenir un facteur incomplet. L'exsitence d'une factorisation de Cholesky incomplète pour une matrice symétrique définie positive n'est pas toujours garantie. Il faut alors utiliser l'option de compensation de la diagonale (`opts.diagcomp`).

5.7 Méthodes itératives

Plusieurs méthodes itératives de résolution de systèmes linéaires sont implémentées dans MATLAB. Quelques unes sont présentées dans le tableau 5.1.

On a choisi de ne présenter que deux méthodes itératives: la méthode du gradient conjugué préconditionné (`pcg`) et la méthode du résidu minimum généralisé (`gmres`).

5.7.1 Méthode du gradient conjugué préconditionné

Lorsque A est symétrique définie positive, la résolution de (5.2) est équivalente à la minimisation (dans $\mathbb{R}^n$) de la fonction quadratique convexe

$$q(x) = \frac{1}{2}x^T Ax - b^T x.$$

Fonction	Méthode
`bicg`	*Bi-Conjugate Gradient method,* pour matrices quelconques
`bicgstab`	*Bi-Conjugate Gradient STABilized method,* pour matrices quelconques
`cgs`	*Conjugate Gradient Squared method,* pour matrices quelconques
`gmres`	*Generalized Minimum RESidual method,* pour matrices quelconques
`minres`	*MINimum RESidual method,* pour matrices symétriques
`pcg`	*Preconditioned Conjugate Gradient method,* pour matrices symétriques définies positives

TABLE 5.1 – Fonctions de résolution de systèmes linéaires par des méthodes itératives

En effet, le gradient de la fonction q est $\nabla q(x) = Ax - b$, et un point x qui minimise q est solution de (5.2) et inversement. En théorie, la méthode du gradient conjugué converge en, au plus, n itérations. Mais en pratique, le système peut être très mal conditionné ou n très grand. Dans la méthode du gradient conjugué préconditionné, on remplace (5.2) par le système

$$M^{-1/2}AM^{-1/2}y = M^{-1/2}b$$

avec $y = M^{1/2}x$. En pratique, la méthode est implémentée sans calculer explicitement $M^{1/2}$ ou $M^{-1/2}$, voir [12, 8, 15] pour les détails de l'algorithme.

Une syntaxe pour utiliser la fonction `pcg` est

```
[x,flag,res,iter]=pcg(A,b,tol,MaxIt,M1,M2,x0)
```

avec, en entrée:

`A` matrice du système, symétrique définie positive

`b` second membre du système

`tol` précision de la solution, la précision par défaut est 10^{-6}

`MaxIt` nombre maximal d'itérations

`M1,M2` facteurs du préconditionneur `M=M1*M2`

`x0` x^0 (optionnel)

et, en sortie:

`x` solution du système

`flag` indicateur de convergence (voir l'aide en ligne pour les autres valeurs de `flag`)
`flag=0` convergence
`flag=1` nombre maximal d'itérations atteint

`res` résidu relatif, $\| b - Ax \| \| b \|^{-1}$

`iter` nombre d'itérations.

Pour les autres formes d'appel de la fonction `pcg`, voir l'aide en ligne.

Dans l'exemple ci-dessous, on utilise `delsq(numgrid('S',50))` la matrice du Laplacien avec condition aux bords de Dirichlet. C'est une matrice d'ordre 2304 symétrique définie positive. Comme second membre on prend `ones(2304,1)`. Le nombre maximal d'itérations, fixé à 50, est atteint sans convergence lorsqu'on utilise `pcg` sans préconditionnement. Avec une matrice de préconditionnement obtenue avec `ichol`, la convergence est atteinte en moins de 50 itérations.

```
>> A=delsq(numgrid('S',50));
>> n=size(A,1)
n = 2304

>> b=ones(n,1);
>> [x,flag,res,iter]=pcg(A,b,10^-6,50,[],[]);  % sans preconditionnement
>> flag
flag =
      1                                        % pas de convergence

>> iter
iter =
      50                                       % max d'itérations atteint

>> opts.type='nofill';
>> R0=ichol(A,opts);                           % Cholesky sans remplissage
>> [x,flag,res,iter]=pcg(A,b,10^-6,50,R0,R0');
>> flag
flag =
     0                                         % convergence

>> iter
iter =
    34                                         % en 34 itérations

>> opts.type='ict'; opts.droptol=0.001;
>> R1=ichol(A,opts);                           % Cholesky incomplète
>> [x,flag,res,iter]=pcg(A,b,10^-6,50,R1,R1');
>> flag
flag =
     0                                         % convergence

>> iter
iter =
     9                                         % en 9 itérations
```

Dans le cas où des permutations ont été effectuées sur les lignes et les colonnes de la matrice A, il faut en tenir compte.

```
>> A=delsq(numgrid('S',50));
>> n=size(A,1); b=ones(n,1);
>> p=symamd(A);                                % permutations avec symamd
>> opts.type='ict'; opts.droptol=0.001;
>> R=ichol(A(p,p),opts);                       % factorisation incomplète
>> x=zeros(n,1);                               % vecteur solution initiale
>> [x(p),flag,res,iter]=pcg(A(p,p),b(p),10^-6,50,R',R);
>> flag
flag =
      0                                        % convergence
```

```
>> iter
iter =
     6                                                   % en 6 itérations
```

La méthode converge maintenant en 6 itérations seulement. La figure 5.6 montre l'historique du résidu relatif. On a utilisé, pour le tracé, `semilogy(x,y)` pour bien visualiser la forme "quadratique" de la courbe.

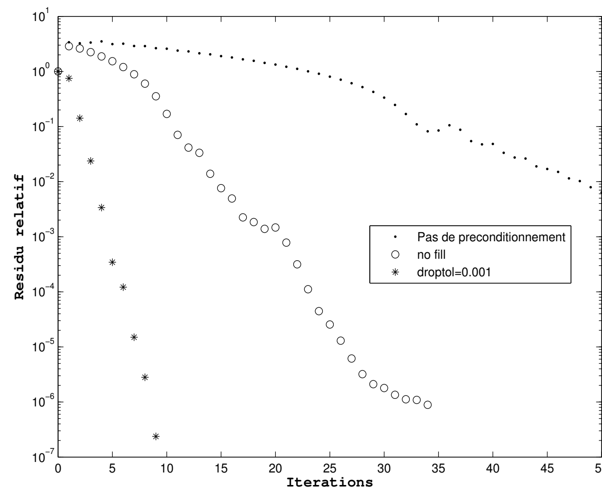

FIGURE 5.6 – Historiques du résidu relatif pour la fonction `pcg` avec ou sans préconditionnement.

5.7.2 Méthode GMRES

Soit x^0 un vecteur de $\mathbb{R}^n$. Le résidu initial du système (5.2) en x^0 est $r^0 = b - Ax^0$. Dans la méthode GMRES, à l'itération k, on détermine x^k qui minimise la norme euclidienne du résidu r^k, c'est-à-dire

$$\| b - Ax^k \|_2 = \min_{v \in W_k} \| b - Av \|_2, \tag{5.6}$$

où le sous-espace affine W_k est donné par

$$W_k = x^0 + \text{vect}\{r^0, Ar^0, \dots, A^{k-1}r^0\}.$$

Le sous-espace vectoriel

$$\text{vect}\{r^0, Ar^0, \dots, A^{k-1}r^0\}$$

est appelé *sous-espace de Krylov* d'ordre k. Le problème (5.6) est un problème de moindres carrés qui nécessite la construction et le stockage d'une base orthogonale du sous-espace $\text{vect}\{r^0, Ar^0, \dots, A^{k-1}r^0\}$. Le coût des itérations et la taille mémoire utilisée augmente donc au fil des itérations. En pratique, on se fixe une dimension maximale m ($10 \leq m \leq 100$) pour le sous-espace de Krylov. Si la convergence n'est pas atteinte au bout de m itérations, on réinitialise l'algorithme en posant $r^0 = r^m$. Le préconditionnement consiste à remplacer l'espace de Krylov par

$$\text{vect}\left\{r^0, (M^{-1}A)r^0, \dots, (M^{-1}A)^{k-1}r^0\right\}.$$

On renvoie à [8, 15, 19] pour les démonstrations et les détails de l'algorithme GMRES.

Une syntaxe pour utiliser la fonction `gmres` est

```
[x,flag,res,iter]=gmres(A,b,restart,tol,MaxIt,M1,M2,x0)
```

avec, en entrée

`A` matrice du système;

`b` second membre du système;

`restart` dimension maximale du sous-espace de Krylov à construire

`tol` précision de la solution, la précision par défaut est 10^{-6};

`MaxIt` nombre maximal d'itérations;

`M1`,`M2` facteurs du préconditionneur `M=M1*M2`;

`x0` x^0, par défaut $x^0 = 0$.

Les arguments résultats ont la même signification que dans la méthode du gradient conjugué sauf `iter` qui est ici un vecteur de taille deux avec $0 \leq \texttt{iter}(1) \leq \texttt{MaxIt}$ et $0 \leq \texttt{iter}(2) \leq \texttt{restart}$.

Dans l'exemple ci-dessous, on résout le système (5.2) avec la matrice `west0479`. On remarque que sans préconditionnement, la méthode GMRES ne converge pas (`flag=1`, nombre maximum d'itérations atteint) au bout de 50 itérations. Avec une matrice de préconditionnement obtenue avec `ilu`, la méthode GMRES converge en 4 itérations.

```
>> load west0479; A=West0479;                    % chargement de la matrice
>> n=size(A,1); b=sum(A,2);                      % second membre tel que x=1
>> [x,flag,res,iter]=gmres(A,b,10,10^-6,50,[],[]);
>> flag
flag =                                           % nombre max d'itérations
     1                                           % atteint

>> setup.droptol=1e-6; setup.type='ilutp';
>> [Li,Ui]=ilu(A,setup);                         % factorisation LU incomplète
>> [x,flag,res,iter]=gmres(A,b,10,10^-6,50,Li,Ui);
>> flag
flag =
     0                                           % convergence

>> iter                                         % en 4 itérations
iter =
     1     4

>> res                                          % résidu final
res =
   1.7947e-08
```

Si des permutations on été effectuées sur la matrice, on en tient compte. Avec la matrice et le second membre du système ci-dessus, le code devient.

```
>> p=symrcm(A);                                   % permutations avec symrcm
>> [Li,Ui]=luinc(A(p,p),setup);                   % factorisation incomplète
>> x=zeros(n,1);                                  % vecteur solution initiale
>> [x(p),flag,res,iter]=gmres(A(p,p),b(p),10,10^-6,50,Li,Ui);
>> flag
flag =
     0                                            %convergence

>> iter
iter =
     1     5                                      % en 5 itérations

>> res
res =
   9.7471e-08                                     % résidu final
```

5.8 Exercices

Exercice 5.1 Générer la version creuse de la matrice tridiagonale R de l'exercice 3.1.

Exercice 5.2 Soit A une matrice carrée (creuse) d'ordre n et s une vecteur d'indices. Comment transformer les lignes s de A en lignes de la matrice identité d'ordre n.

Exercice 5.3 Soit A une matrice carrée (creuse) d'ordre n et s un vecteur d'indices de taille n_s. On souhaite modifier les éléments diagonaux $A(s(i), s(i))$, $i = 1, \dots, n_s$, en les remplaçant par une constante K. Ecrire les instructions MATLAB qui réalise cette transformation.

Exercice 5.4 La fonction MATLAB `colperm` retourne un vecteur de permutation qui classe les colonnes d'une matrice creuse dans un ordre non décroissant d'éléments non nuls.

Considérons la matrice $S = W^T W$ où W est la matrice `west0479` vue au § 5.2. Comme W est non singulière, la matrice S est symétrique définie positive. Calculer la factorisation de Cholesky de la matrice S dans les cas suivants:

- sans permutation de lignes et de colonnes;
- permutation de lignes et colonnes avec `symrcm`;
- permutation de colonnes avec `colperm`;
- permutation de lignes et colonnes avec `symamd`.

Calculer le temps de CPU (avec `tic` et `toc`) de la factorisation ainsi que le nombre d'éléments non nuls du facteur de Cholesky dans chaque cas.

Exercice 5.5 Considérons la matrice tridiagonale par blocs

$$A = \begin{bmatrix} 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 4 & 0 & 0 & -1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 4 & -1 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & -1 & 4 & -1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & -1 & 4 & 0 & 0 & -1 \\ . & . & . & . & . & . & . & . & . \\ 0 & 0 & 0 & -1 & 0 & 0 & 4 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 \end{bmatrix}$$

La matrice A est de la forme

$$A = \begin{bmatrix} T & -I_3 & & & \\ -I_3 & T & -I_3 & & \\ & \ddots & \ddots & \ddots & \\ & & -I_3 & T & -I_3 \\ & & & -I_3 & T \end{bmatrix}$$

où I_3 est la matrice identité d'ordre 3 et T la matrice tridiagonale

$$T = \begin{bmatrix} 4 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 4 \end{bmatrix}.$$

Dans le système d'indexation MATLAB, les diagonales non nulles de A ont pour indices -3, -1, 0, 1 et 3. On souhaite créer la matrice A à l'aide de la fonction `spdiags` en utilisant la forme

```
A=spdiags(B,d,n,n)
```

où `B` est la matrice contenant les colonnes à placer suivant les diagonales spécifiées par le vecteur `d`. Si une colonne de `B` est plus longue que la diagonale qu'elle représente, les éléments d'une sur-diagonale correspondent à la partie inférieure de la colonne de `B`, tandis que les éléments de la sous-diagonale correspondent à la partie supérieure de la colonne de `B`. En d'autres termes, le remplissage des sous-diagonales se fait dans le sens direct, *i.e.* en commençant par le début de la colonne de `B`, tandis que le remplissage des sur-diagonale se fait dans le sens inverse. Il faut donc compléter judicieusement les colonnes représentant les sur- ou sous-diagonales par des zéros.

1. Créer une matrice B, de taille $n \times 5$, qui contient les diagonales non nulles de A.
2. Générer la matrice A avec la fonction `spdiags` pour une taille n quelconque (n multiple de 3).
3. En considérons le second membre `b=ones(n,1)`, résoudre le système $Ax = b$ en utilisant la méthode du gradient conjugué préconditionné `pcg`.

CHAPITRE 6

PROGRAMMATION AVEC MATLAB

MATLAB est aussi un véritable langage de programmation dans lequel on retrouve la plupart des concepts de programmation (fonctions, structures de contrôle, entrée/sortie, ...). Un fichier de programme MATLAB doit obligatoirement avoir l'extension `.m` (d'où le nom de *m–files* donné aux fichiers MATLAB dans la littérature MATLAB). Il y a deux types de fichiers `.m`: les *scripts* et les *fonctions*.

6.1 Types de données et variables

Comme signalé au chapitre 2, toute variable (scalaire, vecteur ou matrice) de MATLAB est un tableau d'éléments. Il n'y a pas de déclarations de variables. Les variables sont déclarées au fur et à mesure de leur apparition. Le type et la dimension d'une variable sont déterminés automatiquement en fonction de la valeur affectée à la variable. On distingue quatre types scalaires: réel, complexe, chaîne et logique.

Le type complexe s'obtient en utilisant l'unité imaginaire `i` ou `j` (i.e. $\sqrt{-1}$).

```
>> z1=2+3i
z1 =
   2.0000 + 3.0000i
>> z2=1+pi*j
z2 =
   1.0000 + 3.1416i
```

On peut aussi écrire un nombre complexe sous sa forme polaire (*ie.* $e^{i\theta}$)

```
>> z=exp(i*pi/6)
z =
   0.8660 + 0.5000i
```

Les fonctions suivantes sont associées au type complexe:

`imag(z)` partie imaginaire de `z`;

`real(z)` partie réelle de `z`;

`abs(z)` module de `z`;

`angle(z)` argument de `z`.

Le type chaîne est en réalité un tableau de caractères. Une donnée de type chaîne est une suite de caractères encadrés par des apostrophes (`'`). Si la chaîne contient une apostrophe, celle-ci doit être doublée.

```
>> ch='Hello'
ch =
    Hello

>> s='Ajourd''hui'
s =
   Ajourd'hui
```

Le type logique ne comprend que deux valeurs: 1 (vrai) et 0 (faux).

```
>> L=pi>2
L =
     1
>> isreal(exp(i*pi/2))
ans =
     0
```

Comme il n'y a pas de déclarations de variables, les fonctions logiques `ischar`, `islogical` et `isreal` permettent de connaître le type d'une variable.

6.2 Structures de données avancées

Les tableaux sont utiles pour stocker des données de même type. Parfois, il est plus utile de rassembler des éléments de type différent au sein d'une entité repérée par un seul nom de variable.

6.2.1 Tableaux de cellules (cell arrays)

Une cellule est une structure MATLAB pouvant contenir toute sorte d'objets (nombre, tableau, chaîne ou même cellules). Un tableau de cellules est donc une collection de variables de nature très différentes. On forme un tableau de cellule avec des accolades.

```
>> c={pi, [1 2 3], 'isima'}
c =
    [3.1416]    [1x3 double]    'isima'
```

Le tableau de cellules `c` ci-dessus contient un double (π), un vecteur 1×3 et une chaine de caractères. L'indexation se fait avec les parenthèse "()" pour récupérer la cellule ou avec des accolades "{}" pour récupérer le contenu de la cellule (*i.e.* sa valeur).

```
>> c(1)
ans =
    [3.1416]

>> c{1}
ans =
    3.1416
```

On remarque que `c(1)` est la cellule contenant π tandis que `c{1}` est la valeur de π.

6.2.2 Structures

Un autre moyen pour regrouper des variables de type différent est la structure. A la différence des tableaux, chaque objet contenu dans la structure (appelé *champ*) a son propre nom.

```
>> clear s          % on s'assure que la structure à créer n'existe pas
>> s.p=rand(20,2);
>> s.t=randn(20,3);
>> s.type='triangle';
```

Comme pour les tableaux, le nombre de champs n'est pas fixe. Ils sont ajoutés à la structure au fur et à mesure de leur apparition dans le code. Le `clear s` ci-dessus permet de d'assurer que la structure `s` n'est pas en mémoire. Si `s` existe et est de type structure, alors l'ancienne structure sera augmentée avec les nouveaux champs.

6.3 Scripts

Un script est un fichier d'instructions MATLAB qui ne comporte pas d'arguments (données ou résultats). En général, un script s'utilise comme un programme principal. Si le script est écrit dans le fichier `monscript.m`, pour l'exécuter il suffit de taper `monscript` (sans extension !) dans la fenêtre de commandes MATLAB. Par exemple, pour tracer la surface définie par un ensemble de points 4.11, on peut utiliser le script de contenu

```
x=rand(100,1)*4-2;
y=rand(100,1)*4-2; z=x.*exp(-x.*x-y.*y);
xu=linspace(min(x),max(x),50);
xu=linspace(min(y),max(y),50);
[X,Y]=meshgrid(xu,yu);
Z=griddata(x,y,z,X,Y,'cubic');
mesh(X,Y,Z)
grid off, axis tight
hold on
plot3(x,y,z,'.')
```

L'avantage ici est qu'en cas de changement de données ou de surface, il suffit de modifier les lignes correspondantes. Attention, un script utilise ou crée des variables dans l'espace de travail.

6.4 Fonctions

Les fichiers de fonctions permettent de créer des fonctions que ne possèdent pas MATLAB. Ces fonctions créées par l'utilisateur auront le même statut que les fonctions internes de MATLAB. La première ligne d'un fichier contenant une fonction MATLAB doit commencer par le mot-clé `function`. Voici comment définir une fonction `mafonct` contenue dans le fichier `mafonct.m`

```
function [var_1,var_2,...,var_n]=mafonct(arg_1,arg_2,...,arg_m)
%MAFONCT informations essentielles
% commentaires pour l'aide en ligne
%
  ...
```

```
instructions
...
```

où

- `var_1,var_2,...,var_n` sont les arguments résultats, *i.e.* les variables de sortie;
- `arg_1,arg_2,...,arg_m` sont les arguments donnés, *i.e.* les variables d'entrée.

Les lignes de commentaires juste après le mot-clé `function` sont affichées lors de la demande d'aide en ligne avec `help`. La première ligne de commentaires est celle qui est scrutée par la commande `lookfor`. Elle doit donc contenir les informations essentielles sur la fonction. Il est impératif de ne pas laisser de ligne vide entre le mot-clé `function` et la première ligne de commentaire.

Il n'y a pas de mot-clé signalant la fin de la fonction. C'est donc la fin du fichier qui indique la fin de la fonction. Toutefois, la commande `return` permet un retour prématuré au programme appelant.

Il est impératif que la fonction et le fichier qui le contient aient le même nom sinon la fonction ne sera pas «visible» par MATLAB. S'il n'y a qu'un seul argument résultat, les crochets sont inutiles. S'il n'y a pas d'arguments résultats (c'est par exemple le cas d'une fonction qui ne fait qu'imprimer ou sauvegarder des données), on utilise les crochets vide

```
function []=imprimer(x)
```

ou on ne met rien dans la partie résultats

```
function imprimer(x)
```

La projection orthogonale v' d'un vecteur v sur un vecteur non nul u est donnée par

$$v' = \frac{u^T v}{\parallel u \parallel^2} u. \tag{6.1}$$

Voici la fonction `proj0` qui calcule la projection d'un vecteur v sur un vecteur non nul u

```
function p=proj0(u,v)
% PROJ0 Projection orthogonale sur un vecteur non nul
% p=proj0(u,v) projection orthogonale de v sur u
% u,v - vecteurs colonne de même taille

p=(u'*v)*u/(u'*u);
```

Pour l'utiliser, il suffit de taper

```
>> u=rand(3,1)
u =
   0.9501
   0.2311
   0.6068

>> v=ones(3,1)
v =
   1
   1
   1
```

```
>> p=proj0(u,v)
p =
   1.2828
   0.3121
   0.8193
```

Pour l'aide en ligne sur la fonction `proj0`

```
>> help proj0
  PROJ0 Projection orthogonale sur un vecteur non nul
  p=proj0(u,v) projection orthogonale de v sur u
  u,v - vecteurs colonne de même taille
```

Si on recherche une fonction MATLAB qui réalise la projection orthogonale

```
 PROJ0 Projection orthogonale sur un vecteur non nul
CAMPROJ Camera projection.
FAN2PARA Convert fan-beam projections to parallel-beam.
PARA2FAN Convert parallel-beam projections to fan-beam.
ipexreconstruct.m: %% Reconstructing an Image from Projection Data
fanUtils Utility functions for fanbeam projections.
```

Les variables d'une fonctions sont locales à la fonction. Un changement de valeur d'un argument d'entrée, dans la fonction, n'est donc pas répercuté dans l'espace de travail MATLAB. Toutefois, il est possible de définir des variables globales avec la commande `global`.

Il existe deux variables prédéfinies `nargin` et `nargout` qui donnent respectivement le nombre d'arguments d'entrée et de sortie. En consultant le contenu de ces variables, on peut faire varier le nombre d'arguments d'entrée ou de sortie d'une fonction. Dans le cas de la fonction `proj0`, on peut la modifier pour qu'elle retourne

- le vecteur projeté sur u, s'il n'y a qu'un seul argument de sortie;
- le vecteur projeté sur $u^{\perp}$, s'il y a deux arguments de sortie.

La fonction devient donc.

```
function [p,pp]=proj(u,v)
% PROJ Projection orthogonale sur un vecteur non nul
% [p,pp]=proj(u,v) projection orthogonale de v
% p=proj(u,v)
% [p,pp]=proj(u,v)
% u,v - vecteurs colonne de même taille

p=(u'*v)*u/(u'*u);
if ( nargout==2)
    pp=v-p;
end
```

On peut maintenant appeler cette nouvelle fonction avec 1 ou 2 arguments de sortie.

```
>> u=rand(3,1)*5, v=ones(3,1)
u =
   4.7506
   1.1557
   3.0342

v =
   1
   1
   1

>> p=proj(u,v)
p =
   1.2828
   0.3121
   0.8193

>> [p,pp]=proj(u,v)
p =
   1.2828
   0.3121
   0.8193

pp =
   -0.2828
    0.6879
    0.1807

>> p'*pp                                       % p orthogonal à pp
ans =
     1.6653e-16
```

Remarque 6.1 *Les arguments dont la valeur est modifiée dans une fonction sont passées par valeur. Les autres sont passés par référence pour économiser la mémoire.*

Il est possible, dans MATLAB de définir une variable qui contient les informations relatives à une fonction (*function handle* en langage MATLAB). Cette variable peut ensuite être passée, en argument, à une autre fonction comme une variable ordinaire. Pour construire une telle variable, il suffit de placer le symbole `@` devant le nom de la fonction (sans espace).

La fonction MATLAB `quad` calcule la valeur approchée de l'intégrale d'une fonction, en utilisant la formule de Simpson de manière adaptative. Elle admet donc comme argument la fonction intégrande.

```
>> help quad
 QUAD   Numerically evaluate integral, adaptive Simpson quadrature.
    Q = QUAD(FUN,A,B) tries to approximate the integral of scalar-valued
    function FUN from A to B to within an error of 1.e-6 using recursive
    adaptive Simpson quadrature. FUN is a function handle. The function
    Y=FUN(X) should accept a vector argument X and return a vector result
    Y, the integrand evaluated at each element of X.
```

```
    Q = QUAD(FUN,A,B,TOL) uses an absolute error tolerance of TOL
    instead of the default, which is 1.e-6.  Larger values of TOL
    result in fewer function evaluations and faster computation,
    but less accurate results.  The QUAD function in MATLAB 5.3 used
    a less reliable algorithm and a default tolerance of 1.e-3.

    Q = QUAD(FUN,A,B,TOL,TRACE) with non-zero TRACE shows the values
    of [fcnt a b-a Q] during the recursion. Use [] as a placeholder to
    obtain the default value of TOL.

    [Q,FCNT] = QUAD(...) returns the number of function evaluations.

    Use array operators .*, ./ and .^ in the definition of FUN
    so that it can be evaluated with a vector argument.
```

Pour utiliser la fonction `quad`, il suffit donc de définir la variable qui contient les informations sur la fonction intégrande et de la passer comme premier argument. Par exemple, pour calculer

$$\int_0^{\pi/2} \sin x \mathrm{d}x,$$

on procède comme suit

```
>> f=@sin

f =
    @sin

>> s=quad(f,0,pi/2)
s =
    1.0000
```

ou, directement

```
>> s=quad(@sin,0,pi/2)
s =
    1.0000
```

Pour évaluer la fonction cible d'une variable de type *function handle*, on utilise la fonction `feval`. La syntaxe est

```
feval(fct,x1,...,xn)
```

La fonction cible de la variable `fct` est évaluée avec les arguments `x1`,...,`xn`. La fonction `feval` est indispensable dans une fonction ayant, comme arguments, des variables de type *function handle*.

On se propose de calculer une solution approchée de l'équation

$$f(x) = 0, \tag{6.2}$$

où f est une fonction de $\mathbb{R}$ dans $\mathbb{R}$ suffisamment régulière. La méthode Newton-Raphson permet d'approcher une solution de (6.2) à l'aide de la suite suivante

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}, \quad k \geq 0.$$

Une fonction MATLAB qui calcule une solution approchée de (6.2) par la méthode de Newton-Raphson doit donc avoir comme argument la fonction f de l'équation.

```
function [x,iter]=newtonraph(f,x,eps)
%Calcul de la racine d'une équation
% par la methode de Newton-Raphson

[y,dy]=feval(f,x);
iter=0;
while (abs(y)>eps & iter<100)
    iter=iter+1;
    [y,dy]=feval(f,x);
    x=x-y/dy;
end
```

Pour résoudre (6.2) avec $f(x) = x\mathrm{e}^x - 10$ dans l'intervalle $[3/2,\ 2]$, on code la fonction

```
function [f,df]=nfct1(x)
f=x*exp(x)-10;
df=exp(x)+x*exp(x);
```

qui retourne la fonction et sa dérivée. Ensuite, on appelle la fonction `newtonraph` avec la borne inférieure de l'intervalle comme x_0.

```
>> f=@nfct1;
>> [x,iter]=newtonraph(f,1.5,0.0001)
x =
    1.7455

iter =
     4

>> [x,iter]=newtonraph(f,1.5,10^-10)
x =
    1.7455

iter =
     5

>> [fx,dfx]=feval(f,x)          % vérification
fx =
     0

dfx =
   15.7289
```

Remarque 6.2 *La fonction* MATLAB *pour résoudre* (6.2) *est* `fzero`.

Remarque 6.3 *Dans les versions antérieures de* MATLAB *(i.e.≤ 5) le passage d'une fonction en argument se faisait à l'aide d'une variable chaîne contenant le nom de la fonction. L'évaluation à l'intérieur de la fonction se faisant toujours avec la fonction* `feval` *dans les mêmes conditions. Dans le cas de l'exemple ci-dessus, on aurait dû faire* `f='nfct1'` *au lieu de* `f=@nfct1`*. Pour des raisons de compatibilité, le passage d'une fonction sous forme de chaîne contenant son nom est aussi accepté.*

6.5 Structures de contrôle

MATLAB dispose de structures de contrôle suivantes : `for` (boucle avec compteur), `while` (répétitive prétestée), `if` (alternative), `switch` (choix ventilé). Toutes ces structures se terminent par un `end`.

6.5.1 La boucle `for`

La syntaxe de la répétitive avec compteur de MATLAB est la suivante.

```
for compteur=debut:pas:fin
 ...
end
```

Les constantes réelles `début` et `fin` sont les bornes inf et sup de l'intervalle à parcourir. La constante (positive ou négative) `pas` est l'incrément, qui vaut 1 par défaut. L'exécution de la boucle se termine lorsque la variable `compteur` devient plus grande que `fin` (cas d'un incrément positif), ou plus petite que `debut` (cas d'un incrément négatif).

Le bout de code (en ligne!) suivant calcule, à l'aide de la boucle `for`, le n-ème terme de la suite de Fibonacci ($u_0 = u_1 = 1$)

$$u_n = u_{n-1} + u_{n-2}, \quad n \geq 2.$$

```
>> n=20;
>> u0=1; u1=1;
>> for i=2:n, u=u1+u0; u0=u1; u1=u; end
>> u
u =
       10946
```

Les boucles `for` peuvent être imbriquées. Par exemple, la matrice de Hilbert de taille $m \times n$ peut-être obtenue à l'aide de a double boucle suivante

```
for i = 1:m
    for j = 1:n
        H(i,j)=1/(i+j-1);
    end
end
```

6.5.2 La boucle `while`

Très utile dans les algorithmes itératifs, la boucle `while` consiste à exécuter une séquence d'instruction tant qu'une condition est vérifiée. La syntaxe de la boucle `while` est

```
while expression_logique
 ...
end
```

où `expression_logique` est en général un test. Pour éviter une boucle sans fin, les instructions dans le corps de la boucle doivent impérativement agir sur le résultat de `expression_logique`.

La boucle suivante calcule les termes successifs de la suite de Fibonacci et s'arrête si $u_n > 10^6$.

```
u0=1; u1=1; u=2; n=3;

while (u < 10^6)
      n=n+1;
      u=u1+u0;
      u0=u1; u1=u;
end
```

6.5.3 L'alternative **`if`**

L'alternative simple a pour syntaxe

```
if expression_logique
   instructions I
else
   instructions II
end
```

La séquence d'instructions `instructions I` est exécutée si `expression_logique` est vraie, sinon c'est la séquence d'instructions `instructions I` qui est exécutée. A noter que la partie `else` peut-être facultative. Dans ce cas l'alternative se réduit à

```
if expression_logique
   instructions I
end
```

L'alternative complète a pour syntaxe

```
if expression_logique 1
   instructions I
elseif expression_logique 2
   instructions II
   ...
elseif expression_logique n
   instructions N
else
   instructions par défaut
end
```

Le mot-clé est bien `elseif` et non `else if` (il n'y a pas d'espace!)

6.5.4 Les expressions logiques

Comme dans les autres langages, les expressions logiques sont obtenues dans MATLAB avec les opérateurs de comparaison, les opérateurs et les fonctions logiques. Les opérateurs de comparaison de MATLAB sont fournis dans le tableau 6.1.

Opérateur	signification
<	inférieur à
<=	inférieur ou égal à
>	supérieur à
>=	supérieur ou égal à
==	égal à
~=	différent de

TABLE 6.1 – Opérateurs de comparaison de MATLAB

Les opérateurs de comparaison du tableau 6.1 peuvent s'appliquer aussi aux tableaux. Les opérandes doivent être de même dimension ou l'un des deux doit être un scalaire. Le résultat d'une opération de comparaison sur des tableaux est un tableau de `0` (là où la relation est fausse) et de `1` (là où la relation est vraie).

```
>> A=[2 3 8; 10 7 2; 3 4 8];
>> B=[2 10 9; 2 7 2; 3 5 7];
>> A==B
ans =
     1     0     0
     0     1     1
     1     0     0

>> A>B
ans =
     0     0     0
     1     0     0
     0     0     1
```

Si l'un des opérandes matriciels est vide (*i.e.* matrice `[]`), l'autre doit être une matrice vide ou un scalaire. Dans le cas contraire, on a un message d'erreur.

Les opérateurs logiques de MATLAB sont fournis dans le tableau 6.2

Les opérateurs logiques de MATLAB s'appliquent aussi aux tableaux. Les opérandes doivent être de même dimension ou l'un des deux doit être un scalaire. Le résultat est un tableau de `0` et de `1`, chaque opérateur ayant sa propre règle:

- `&` le résultat est `1` si les deux opérandes sont non nuls, `0` sinon;
- `|` le résultat est `1` si l'un des deux opérandes est non nul, `0` sinon;
- `~` le résultat est `1` si l'opérande est nul, `0` sinon.

L'exemple suivant illustre comment ça marche.

Opérateur	Signification
&	*et* logique
|	*ou* logique
~	négation

TABLE 6.2 – Opérateurs logiques de MATLAB

```
>> A=[0 1 7 2 0 6];
>> B=[1 3 8 0 9 5];
>> A & B
ans =
     0     1     1     0     0     1

>> A | B
ans =
     1     1     1     1     1     1

>> ~A
ans =
     1     0     0     0     1     0
```

6.5.5 Instructions de rupture de séquence

Instruction	Signification
`break`	sortie anticipée d'une boucle
`continue`	bouclage anticipée
`pause`	arrêt momentané
`return`	sortie anticipée d'une fonction
`error`	interruption du programme avec message d'erreur

TABLE 6.3 – Instructions de rupture de séquence

Il existe dans MATLAB plusieurs instructions de rupture de séquences, résumées dans le Tableau 6.3.

La commande `break` termine l'exécution d'une boucle `for` ou `while`. Dans le cas de boucles imbriquées, `break` permet de sortir de la boucle la plus proche.

Pour passer à l'itération suivante, dans une boucle `for` ou `while`, on utilise l'instruction `continue`.

La commande `return` termine l'exécution de la fonction ou du script en cours et permet de revenir au ficher appelant ou au clavier.

La commande `pause` interrompt momentanément l'exécution du programme. Il existe deux variantes:

- `pause` attend que l'utilisateur appuie sur une touche du clavier pour continuer l'exécution du programme;
- `pause(n)` le programme s'arrête pendant `n` secondes avant de se poursuivre.

L'activation ou la désactivation de toutes les commandes `pause` d'un programme MATLAB se fait avec `pause on` ou `pause off`.

Enfin la commande `error('message')` affiche le message spécifié, émet un «bip» et interrompt l'exécution du programme.

6.6 Quelques fonctions internes

MATLAB dispose de plusieurs fonctions qui permettent le plus souvent d'éviter l'emploi de boucle pour parcourir un tableau.

6.6.1 Les fonctions logiques

Fonction	Signification
`all`	1 si tous les éléments d'un tableau sont non nuls
`any`	1 si l'un des éléments d'un tableau est non nul
`isnan`	détecte les `NaN`
`isinf`	détecte les `Inf`
`isfinite`	détecte les éléments finis (≠`NaN`, `Inf`)

TABLE 6.4 – Fonctions logiques

En plus des opérateurs logiques, MATLAB dispose de plusieurs fonctions logiques, tableau 6.4.

- `all(A)` retourne `1` si tous les éléments du vecteur `A` sont vrais (*i.e.* non nuls), `0` sinon. Si `A` est une matrice, la fonction s'applique sur les colonnes et la valeur retournée est un vecteur de dimension le nombre de colonnes de `A`.

```
>> A=[0 2 5; 6 2 2]
A =
     0     2     5
     6     2     2

>> all(A)
ans =
     0     1     1
```

- `any(A)` retourne `1` si l'un des éléments du vecteur `A` est non nul, `0` sinon. Si `A` est une matrice, la fonction s'applique sur les colonnes et la valeur retournée est un vecteur de dimension le nombre de colonnes de `A`.

```
>>A =
     0     2     5
```

```
     0     2     0

>> any(A)
ans =
     0     1     1
```

- `B=isnan(A)` retourne le tableau `B`, de même taille que `A` tel que `B(i,j)=1` si `A(i,j)=NaN`, `0` sinon.

```
>> A=[1 0/0 6 2]
A =
     1   NaN     6     2

>> isnan(A)
ans =
     0     1     0     0
```

- `B=isinf(A)` comme `isnan` mais détecte les éléments de valeur `Inf` d'un tableau.

```
>> A=[1 1/0 6 2]
A =
     1   Inf     6     2

>> isinf(A)
ans =
     0     1     0     0
```

- `B=isfinite(A)` détecte les éléments finis d'une matrice, *i.e.* `B(i,j)=1` si `A(i,j)` est fini, `B(i,j)=0` si `A(i,j)=NaN` ou `A(i,j)=Inf`.

6.6.2 La fonction `find`

La fonction `find` permet de sélectionner les indices des éléments d'un tableau vérifiant une condition logique. Elle remplace avantageusement une boucle (`for` ou `while`). Dans l'exemple suivant, on sélectionne, parmi un ensemble de points, ceux qui appartiennent au disque unité $\{(x, y) \mid x^2 + y^2 <= 1\}$.

```
>> x=2*rand(1,20)-2;
>> y=2*rand(1,20)-2;
>> i=find(x.*x+y.*y<=1)
i =
   4     7    13    17    18
```

Dans le cas où l'argument d'entrée de la fonction `find` est une matrice, on a le choix des arguments de sortie.

```
>> P=pascal(5)
P =
   1     1     1     1     1
   1     2     3     4     5
   1     3     6    10    15
   1     4    10    20    35
```

```
   1     5    15    35    70

>> [I,J]=find(P>30)
I =
   5
   4
   5

J =
   4
   5
   5

>> IJ=find(P>30)
IJ =
    20
    24
    25
```

Dans le premier cas, les éléments qui vérifient la relation sont

```
P(I(1),J(1))
P(I(2),J(2))
P(I(3),J(3))
```

Dans le deuxième cas, les indices retournés sont ceux du vecteur `P(:)`.

6.6.3 Fonctions de réduction

Les fonctions de réduction sont des fonctions de transformation qui réduisent la taille des tableaux. Nous avons emprunté ce terme (et sa définition) au langage Fortran 90 (voir *e.g.* [5]).

Le tri

La fonction `sort` tri les éléments d'un vecteur passé en argument. Le tri est effectué dans l'ordre croissant.

```
>> x=randperm(6)  %effectue une permutation aléatoire des entiers de 1 à 6
x =
   3     2     4     5     1     6

>> xt=sort(x)
xt =
     1     2     3     4     5     6
```

Un deuxième argument de sortie permet de récupérer les indices triés. Avec le vecteur `x` ci-dessus, on a.

```
>> [xt,it]=sort(x)
xt =
    1     2     3     4     5     6

it =
    5     2     1     3     4     6
```

Si l'argument est une matrice, le tri est effectué colonne par colonne, de manière indépendante. Un deuxième argument permet de spécifier la dimension à considérer pour le tri.

```
>> H=hilb(4)                              % tri par colonnes
H =
   1.0000    0.5000    0.3333    0.2500
   0.5000    0.3333    0.2500    0.2000
   0.3333    0.2500    0.2000    0.1667
   0.2500    0.2000    0.1667    0.1429

>> [Ht,It]=sort(H)
Ht =
    0.2500    0.2000    0.1667    0.1429
    0.3333    0.2500    0.2000    0.1667
    0.5000    0.3333    0.2500    0.2000
    1.0000    0.5000    0.3333    0.2500

It =
    4     4     4     4
    3     3     3     3
    2     2     2     2
    1     1     1     1

>> [H1,I1]=sort(H,2)                     % tri par ligne
H1 =
    0.2500    0.3333    0.5000    1.0000
    0.2000    0.2500    0.3333    0.5000
    0.1667    0.2000    0.2500    0.3333
    0.1429    0.1667    0.2000    0.2500

I1 =
    4     3     2     1
    4     3     2     1
    4     3     2     1
    4     3     2     1
```

Les fonctions `min` et `max`

Les fonctions `min` et `max` retournent l'élément minimum ou maximum d'un vecteur passé en argument et, optionnellement, l'indice qui réalise le minimum ou le maximum.

```
>> x=rand(1,5)
x =
    0.1197    0.0381    0.4586    0.8699    0.9342

>> m=min(x), M=max(x)
m =
   0.0381

M =
   0.9342

>> [m,im]=min(x)
m =
   0.0381

im =
    2
```

Lorsque l'argument d'entrée est une matrice, la recherche du minimum ou du maximum se fait colonne par colonne. Les fonctions retournent alors un vecteur ligne des minima ou maxima sur chaque colonne. Le deuxième argument de sortie (optionnel) est aussi un vecteur qui contient les indices des lignes qui réalisent le minimum ou le maximum, pour chaque colonne.

```
>> A=round(10*rand(5))
A =
   5     0     4     0     8
   4     7     2     4     4
   9     7     6     8     6
   0    10     7     8     7
   3     6     4     9     2

>> [M,I]=max(A)
M =
   9    10     7     9     8

I =
   3     4     4     5     1
```

La variante à deux arguments d'entrée permet de sélectionner le plus grand élément des deux arguments.

```
>> x=randperm(10)
x =
   2     5     8     7     4    10     9     6     1     3

>> y=round(rand(1,10)*20)
y =
   16    19    10    18     3    20     5     5    18    15
```

```
>> xymax=max(x,y)
xymax =
      16    19    10    18     4    20     9     6    18    15

>> xymin=min(x,y)
xymin =
       2     5     8     7     3    10     5     5     1     3
```

Comme pour la fonction `sort`, les variantes `min(A,[],dim)` ou `max(A,[],dim)` permettent de spécifier la dimension de recherche.

```
>> A=round(rand(4)*10)
A =
   1     3     1     5
   0     7    10     3
   9     3     6     4
   2     5     4     2

>> max(A,[],1)                                 % max sur les colonnes
ans =
     9     7    10     5

>> max(A,[],2)                                 % max sur les lignes
ans =
     5
    10
     9
     5
```

Pour avoir l'élément minimum ou maximum d'une matrice `A`, il suffit de faire la recherche sur le vecteur correspondant `A(:)`

```
>> A=round(10*randn(5))
A =
   -1     4    -3     3    -4
    4    -9   -12     2   -12
    1     8   -22     0   -11
   -6     6    10   -10    15
   -6    -8    -5    -9     1

>> m=min(A(:)), M=max(A(:))
m =
   -22

M =
   15
```

Les fonctions `sum` et `prod`

Les fonctions `sum` et `prod` effectuent la somme et le produit des éléments d'un tableau passé en argument. Les règles pour une matrice sont les mêmes que pour les fonction `min` et `max`. La fonction est appliquée sur les colonnes de la matrice passée en argument à moins qu'un deuxième argument optionnel ne précise la dimension de travail.

```
>> x=randperm(10)
x =
   5     6     9     3    10     1     4     8     2     7

>> sx=sum(x), px=prod(x)      % somme et produit des éléments de x
sx =
    55

px =
    3628800

>> A=round(rand(4)*10)
A =
   8     4     8     4
   1     3    10     5
   6     9    10     2
   1     0     8     6

>> sum(A)                     % somme sur les colonnes de A (par défaut)
ans =
     16    16    36    17

>> sum(A,2)                   % somme sur les lignes de A
ans =
     24
     19
     27
     15

>> prod(A,1)                  % produit sur les colonnes de A
ans =
     48            0        6400         240
```

6.6.4 Opérations sur les ensembles

MATLAB dispose de fonctions spécialisées effectuant les opérations sur les ensembles telles que l'union, l'intersection, le complémentaire, etc.

La fonction `union` réalise la réunion de deux vecteurs considérés comme des ensembles. Le vecteur résultat est trié.

```
>> x=union([2 6 1 0],[1 7 3 2 10 5])
x =
```

```
   0     1     2     3     5     6     7    10

>> x=union([2 6 1 0 2 2],[1 7 3 2 10 5],'rows')
x =
   1     7     3     2    10     5
   2     6     1     0     2     2
```

Dans la deuxième forme (*i.e.* avec l'option `'rows'`) la fonction `union` retourne une combinaison de ligne des arguments d'entrée sans répétition. Dans ce cas les deux arguments doivent avoir le même nombre de colonnes. Avec l'argument optionnel `'rows'`, la fonction `union` accepte aussi des matrices en arguments (avec le même nombre de colonnes).

```
>> A=round(10*rand(3,4))
A =
   9     2     3     6
   6     5     7     7
   6     9     4     4

>> P=pascal(4)
P =
   1     1     1     1
   1     2     3     4
   1     3     6    10
   1     4    10    20

>> AP=union(A,P,'rows')
AP =
     1     1     1     1
     1     2     3     4
     1     3     6    10
     1     4    10    20
     6     5     7     7
     6     9     4     4
     9     2     3     6
```

Des arguments de sortie optionnels permettent de récupérer les indices des éléments qui réalisent la réunion (le résultat final est trié!).

La fonction `intersect` réalise l'intersection de deux vecteurs ou de deux matrices, considérés comme des ensembles, dans les mêmes conditions (et les mêmes options) que la fonction `union`. Le résultat peut être un ensemble vide. Avec les matrices `A` et `P` utilisées pour `union` on a

```
>> x=intersect(A(3,:),P(3,:))
x =
   6

>> C=intersect(A,P,'rows')
C =
   Empty matrix: 0-by-4
```

La fonction `setdiff(x,y)` retourne le complémentaire du vecteur `y` dans le vecteur `x`. En d'autres termes, la fonction `setdiff(x,y)` retourne les éléments de `x` qui ne sont pas dans `y`. Avec deux matrices ayant le même nombre de colonnes, la fonction retourne les lignes de `x` qui ne sont pas dans `y`.

```
>> x=setdiff([2 8 3 7 4],[3 2 1])
x =
   4     7     8

>> A=[ 2 4 3; 1 0 2; 0 3 4], B=[1 1 1; 2 4 3]
A =
   2     4     3
   1     0     2
   0     3     4

B =
   1     1     1
   2     4     3

>> C=setdiff(A,B,'rows')
C =
   0     3     4
   1     0     2
```

La fonction `setxor(x,y)` retourne les éléments qui ne sont pas dans l'intersection de `x` et `y` dans les mêmes conditions (et options) que les fonctions précédentes.

La fonction `ismember(x,S)` retourne 1 si `x` appartient à la matrice `S` et 0 sinon. Si `x` est un vecteur, retourne un vecteur `y` de même longueur que `x`. A chaque composante de `x` qui appartient à `S`, la composante correspondante de `y` vaut 1 et 0 sinon.

```
>> A=pascal(3)
A =
   1     1     1
   1     2     3
   1     3     6

>> ismember(1,A)
ans =
     1

>> ix=ismember([1 2 3 4],A)
ix =
    1     1     1     0
```

Avec deux matrices ayant le même nombre de colonnes et l'option `'rows'`, le test d'appartenance se fait sur les lignes. Dans l'exemple suivant, `A` est la matrice de Pascal utilisée ci-dessus.

```
>> B=[2 6 8; 1 3 6; 1 2 3]
B =
   2     6     8
   1     3     6
```

```
   1     2     3

>> ismember(B,A,'rows')
ans =
     0
     1
     1
```

La première ligne de la matrice B n'appartient pas à la matrice de Pascal d'ordre 3. D'où le 0 du vecteur résultat.

6.7 Importation/exportation de données

6.7.1 Les fonctions save et load

On a déjà vu les commandes save et load au chapitre 1 pour la sauvegarde et le chargement de données binaires (fichier .mat). Dans cette section, on ne s'occupe que de la sauvegarde et du chargement de données numériques sous forme de fichier texte.

La forme fonctionnelle de load permet de lire directement dans un tableau des données numériques, à partir d'un fichier texte. Les données doivent être rangées en colonnes et le nombre de colonnes doit être le même pour toutes les lignes du fichier.

Supposons que le fichier fdonnees.dat contienne les lignes suivantes.

```
   -0.4326    0.3273    1.0668
   -1.6656    0.1746    0.0593
    0.1253   -0.1867   -0.0956
    0.2877    0.7258   -0.8323
   -1.1465   -0.5883    0.2944
    1.1909    2.1832   -1.3362
    1.1892   -0.1364    0.7143
   -0.0376    0.1139    1.6236
```

On peut le lire directement et affecter son contenu à la matrice A comme suit.

```
>> clear all
>> A=load('fdonnees.dat');
>> A
A =
   -0.4326    0.3273    1.0668
   -1.6656    0.1746    0.0593
    0.1253   -0.1867   -0.0956
    0.2877    0.7258   -0.8323
   -1.1465   -0.5883    0.2944
    1.1909    2.1832   -1.3362
    1.1892   -0.1364    0.7143
   -0.0376    0.1139    1.6236
```

La commande save peut aussi sauvegarder des variables sous format texte. Il suffit de le préciser en option.

```
>> a=rand(3)
a =
   0.9501    0.4860    0.4565
   0.2311    0.8913    0.0185
   0.6068    0.7621    0.8214

>> b=ones(1,4)
b =
   1     1     1     1

>> save -ascii TDonnees.dat a b    % sauvegarde a et b dans TDonnees.dat
>> type TDonnees.dat               % affiche le contenu du fichier TDonnees.dat

   9.5012929e-01   4.8598247e-01   4.5646767e-01
   2.3113851e-01   8.9129897e-01   1.8503643e-02
   6.0684258e-01   7.6209683e-01   8.2140716e-01
   1.0000000e+00   1.0000000e+00   1.0000000e+00   1.0000000e+00
```

Il existe aussi une forme fonctionnelle de `save` (voir l'aide en ligne).

6.7.2 Les fonctions `fprintf` et `fscanf`

Les fonctions `fprintf` et `fscanf` permettent de lire et écrire dans un fichier formaté. Avant de lire ou écrire dans un fichier, il faut l'ouvrir avec la fonction `fopen`.

```
fid=fopen('NomFichier.dat','permission')
```

`fid` est un entier. On referme le fichier avec la fonction `fclose(fid)`. La chaîne `'permission'` spécifie le mode d'accès au fichier

`'r'` accès en lecture

`'w'` accès en écriture

`'a'` ajout de données

`'r+'` accès en lecture et écriture.

La syntaxe des fonctions `fscanf` et `fprintf` est la suivante.

- **`fprintf(fid,'format',variables)`** écrit les variables au format spécifié par la chaîne de caractères `'format'` dans le fichier désigné par le pointeur de fichier `fid`. Si l'entier `fid` vaut 1 ou s'il est omis, l'écriture se fait à l'écran.
- **`a=fscanf(fid,'format',size)`** assigne à la variable `a` les données lues dans le fichier identifiée par `fid` au format `'format'`. `size` est un vecteur de la forme `[m,n]` qui spécifie la taille de `a`. Attention, chaque ligne du fichier lu est transformée en un colonne de `a`.

Les spécifications de formatage sont celles du langage C. Les spécifications de formatage les plus utilisées (pour nous) sont

`%md` entier sur `m` colonnes

`%m.nf` réel flottant sur `m` colonnes avec `n` décimales

`%m.ne` réel sous forme exponentielle sur `m` colonnes avec `n` décimales.

`\n` retour à la ligne.

Les fonctions `fscanf` et `fprintf` sont vectorisées.

```
>> t=0:.25:1; y=sin(t);
>> fprintf('t=%10.6f  y=%10.6f \n',t,y)
t=  0.000000  y=  0.250000
t=  0.500000  y=  0.750000
t=  1.000000  y=  0.000000
t=  0.247404  y=  0.479426
t=  0.681639  y=  0.841471
```

Supposons que le fichier `donnees.dat` est de la forme

```
1000 256
2888 265
2788 543
7865 122
8754 677
```

Alors on peut le lire avec la suite de commandes.

```
>> fid=fopen('donnees.dat','r');
>> a=fscanf(fid,'%4d%3d',[2,5]);
>> fclose(fid);
>> a
a =
   1000        2888        2788        7865        8754
    256         265         543         122         677

>> a'
ans =
     1000         256
     2888         265
     2788         543
     7865         122
     8754         677
```

L'inconvénient de `fscanf` par rapport à `load` c'est qu'il faut absolument spécifier le format sous peine d'erreur.

6.8 Optimisation d'un code

Comme son nom l'indique, MATLAB est conçu pour le calcul matriciel. Les instructions performantes sont donc celles qui comportent des opérations matricielles ou vectorielles. L'exemple suivant en est une illustration.

```
>> tic, i=0; for t=0:0.01:100, i=i+1; y(i)=sin(t); end, toc
elapsed_time =
              0.2446
```

```
>> tic, t=0:0.01:100; y=sin(t); toc
elapsed_time =
              0.0028
```

La *vectorisation* consiste à transformer une boucle `for` ou `while` en opérations matricielles ou vectorielles équivalentes. Nous allons l'illustrer sur le calcul de la matrice de distances d'un ensemble de points de $\mathbb{R}^2$.

Soit $\mathcal{E} = \{M_i = (x_i, y_i)\}_{i=1,\dots,n}$ un ensemble de points dans $\mathbb{R}^2$. On veut calculer la matrice de distance $\mathcal{E}$, *i.e.* la matrice $D = (d_{ij})$ où

$$d_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}, \quad i = 1, \dots, n.$$

La première approche (directe) est d'utiliser des boucles `for` comme en C/C++ ou en Fortran. Le résultat est la fonction 6.1 (`matd0`).

Fonction 6.1 Fonction matrice de distance avec des boucles `for`

```
function d=matd0(x,y)
%
% Matrice de distances d'un ensemble de points
%
%
n=length(x);
d=zeros(n);

% boucle de calcul des distances
for i=1:n
    for j=1:n
        d(i,j)=sqrt((x(i)-x(j))^2+(y(i)-y(j))^2);
    end
end
```

On remarque la matrice D est la norme de la matrice

$$\begin{bmatrix} M_1M_1 & M_1M_2 & \cdots & M_1M_n \\ M_2M_1 & M_2M_2 & \cdots & M_2M_n \\ \vdots & \vdots & \cdots & \vdots \\ M_nM_1 & M_nM_2 & \cdots & M_nM_n \end{bmatrix} = \begin{bmatrix} M_1 & M_2 & \cdots & M_n \\ M_1 & M_2 & \cdots & M_n \\ \vdots & \vdots & \cdots & \vdots \\ M_1 & M_2 & \cdots & M_n \end{bmatrix} - \begin{bmatrix} M_1 & M_1 & \cdots & M_1 \\ M_2 & M_2 & \cdots & M_2 \\ \vdots & \vdots & \cdots & \vdots \\ M_n & M_n & \cdots & M_n \end{bmatrix}$$

Il suffit donc de répliquer judicieusement les vecteurs $x = (x_i)$ et $y = (y_i)$. C'est ce que fait la fonction 6.2.

On obtient à l'exécution:

```
>> x=rand(5,1)*30-15;
>> y=rand(5,1)*30-15;
>> d=matd(x,y)
d =
         0    8.8501   25.6542    2.5325    2.4329
```

Fonction 6.2 Fonction matrice de distances vectorisée

```
function d=matd(x,y)
%
% Matrice de distances d'un ensemble de points
%
%
n=length(x);

% replication de x et y en n colonnes
xx=x(:,ones(1,n));
yy=y(:,ones(1,n));

% calcul des distances
d=sqrt((xx-xx').^2+(yy-yy').^2);
```

```
    8.8501         0   18.3946   10.5236    6.4193
   25.6542   18.3946         0   26.0717   23.5964
    2.5325   10.5236   26.0717         0    4.4530
    2.4329    6.4193   23.5964    4.4530         0
```

La comparaison des deux fonctions est sans appel.

```
>> x=rand(100,1); y=rand(100,1);
>> tic, d=matd0(x,y); toc
elapsed_time =
              0.1335

>> tic, d=matd(x,y); toc
elapsed_time =
              0.0021

>> x=rand(1000,1); y=rand(1000,1);
>> tic, d=matd0(x,y); toc
elapsed_time =
              11.8824

>> tic, d=matd(x,y); toc
elapsed_time =
              0.3486
```

La fonction 6.2, vectorisée, est environ 30 fois plus rapide que la fonction 6.1, non vectorisée. Ceci même en pré-allouant la matrice de distance `d` avant le début des boucles dans la fonction 6.1.

Un autre moyen pour améliorer un code MATLAB est la gestion de la mémoire. L'allocation dynamique des tableaux peut entraîner une fragmentation de la mémoire. Il y a un risque qu'il n'y ait plus assez d'espace mémoire contigüe pour l'allocation d'un grand tableau. On peut réduire la fragmentation de la mémoire en pré-allouant certains tableaux. Pour les tableaux numériques, il suffit d'utiliser la fonction `zeros(m,n)`. Il faut aussi utiliser la commande `clear` pour libérer de l'espace.

6.9 Exercices

Exercice 6.1 Soit A une matrice $m \times n$. Traduire en MATLAB (sans boucle) les expressions suivantes

$$\prod_{j=1}^{n}\left(\sum_{i=1}^{m}|a_{ij}|\right), \qquad \prod_{i=1}^{m}\left(\sum_{j=1}^{n}|a_{ij}|\right).$$

Exercice 6.2 Traduire l'instruction Matlab

```
max(sqrt(sum([x y].^2,2)))
```

où x et y sont des vecteurs colonnes de longueur n.

Exercice 6.3 Ecrire une fonction MATLAB qui calcule sans boucle le vecteur x de dimension n dont les composantes sont données par

$$x_i = \sum_{j=1}^{m} \log(i)\mathrm{e}^{j^2}, \quad i = 1, \ldots, n.$$

Exercice 6.4 Écrire les fonctions MATLAB vectorielles qui évaluent les fonctions, de $\mathbb{R}^n$ dans $\mathbb{R}$, suivantes:

$$f_1(x) = \sum_{i=2}^{n} i\,(2x_i - x_{i-1})^2,$$

$$f_2(x) = 10^{-5}\sum_{i=1}^{n}(x_i - 1)^2 + \left(\sum_{i=1}^{n} x_i^2 - 0.25\right)^2.$$

Exercice 6.5 (Arêtes d'une triangulation) On suppose qu'on dispose d'une triangulation `(t,p)`, où

— `p(1:np,1:2)` coordonnées des sommets du triangles;

— `t(1:nt,1:3)` est le tableau de connectivité de la triangulation, *i.e.* `t(k,1:3)` contient les numéros des sommets du triangle `k`.

1.– Comment extraire facilement toutes les arêtes de la triangulation ?
2.– En principe les arêtes internes apparaissent deux fois (à une permutation de sommets prêt) tandis que les arêtes externes (du bord) n'apparaissent qu'une seule fois. Comment extraire les arêtes externes ? (*Indication :* utiliser les fonctions `unique`, `sort` et `histc`).

Exercice 6.6 (Méthode de Newton-Raphson) Soit à résoudre le système non linéaire

$$(P) \qquad F(x) = 0,$$

où $F\ :\mathbb{R}^n \longrightarrow \mathbb{R}^n$ est «suffisamment» dérivable. La méthode de Newton-Raphson [6, 17, 18, 20, 19] permet d'approcher une solution de (P) grâce à la formule itérative récurrente

$$\begin{aligned} w^{k+1} &= -\Big[\nabla F(x^k)\Big]^{-1} F(x^k), \quad k \geq 0, \\ x^{k+1} &= x^k + w^k, \end{aligned}$$

où $\nabla F(x^k)$ est la matrice Jacobienne de F, supposée inversible. Donc à chaque itération k, on doit résoudre le système linéaire en w^{k+1}

$$\nabla F(x^k) w^{k+1} = -F(x^k), \quad k \geq 0.$$

On arrête les itérations dès que l'erreur relative sur x^k, *i.e.* $\| w^k \| / \| x^k \|$, devient «suffisamment» petite.

Écrire un script MATLAB pour résoudre, par la méthode de Newton-Raphson, un système d'équations non linéaires. La matrice jacobienne $\nabla F(x)$ et la fonction $F(x)$ devront être calculées dans la même fonction MATLAB. on pourra utiliser le système suivant pour la mise au point du programme.

$$F_1(x) = 7x_1^2 + 3x_1x_2 + 4x_1 - x_2 - 41 = 0, \quad (6.3)$$

$$F_2(x) = 10x_1^2 + 4x_1x_2 + 5x_1 - 2x_2 - 56 = 0. \quad (6.4)$$

On prendra $x^0 = (-1, -3)$ et on limitera le nombre d'itérations à 40. Pour information la solution de (6.3)-(6.3) est $x^* = (2, 1)$. Modifier le script Matlab pour qu'il réalise (optionnellement) la représentation graphique des itérés successifs x^k, lorsqu'on est dans $\mathbb{R}^2$ ou $\mathbb{R}^3$.

Tester la robustesse de votre programme, pour des grandes valeurs de n, avec les fonctions F suivantes

Fonction de Rosenbrock:

$$F_{2i-1}(x) = 10(x_{2i} - x_{2i-1}^2),$$

$$F_{2i}(x) = 1 - x_{2i-1}$$

pour $i = 1, \dots, n/2$ (donc n entier pair), $x_0 = (-1.2, 1, \dots, -1.2, 1)$ et la solution exacte est $x^* = (1, 1, \dots, 1)$.

Fonction trigonométrique:

$$F_i(x) = n - \sum_{j=1}^{n} \left(\cos x_j + i(1 - \cos x_i) - \sin x_i\right),$$

pour $i = 1, \dots, n$, $x_0 = (1/n, \dots, 1/n)$.

Exercice 6.7 (Méthode de Levenberg-Marquard) Pour les problèmes de moindres carrés non linéaires, la méthode de Levenberg-Marquardt est la méthode standard, voir par exemple [2, 14, 18]. Elle combine astucieusement les qualités de deux méthodes d'optimisation: la robustesse de la méthode de la plus forte pente (lorsqu'on est loin de l'optimum) et l'efficacité de la méthode de Newton (au voisinage de l'optimum). On peut consulter [2, 16] pour les détails sur ces méthodes d'optimisation.

Soit un modèle à ajuster $y = y(x; \boldsymbol{a})$, dépendant non linéairement de n paramètres $\boldsymbol{a} = (a_1, \dots, a_n)^T$. On dispose de p mesures (x_i, y_i), $i = 1, \dots, p$. Soit F la fonction, de $\mathbb{R}^n$ dans $\mathbb{R}^p$, définie par

$$F(\boldsymbol{a}) = (F_1(\boldsymbol{a}), \dots, F_p(\boldsymbol{a}))^T$$

avec $F_i(\boldsymbol{a}) = y(x_i; \boldsymbol{a}) - y_i$, pour $i = 1, \dots, p$.

L'erreur, en norme Euclidienne, entre les mesures et le modèle est alors donnée par

$$e(\boldsymbol{a}) = \frac{1}{2} F(\boldsymbol{a})^T F(\boldsymbol{a}) = \frac{1}{2} \| F(\boldsymbol{a}) \|_2^2 .$$

L'ajustement consiste donc à trouver le paramètre $\boldsymbol{a}$ qui minimise la fonction $e(\boldsymbol{a})$, *i.e.* résoudre le problème d'optimisation

$$\min_{\boldsymbol{a} \in \mathbb{R}^n} e(\boldsymbol{a}).$$

On note $J_F(\boldsymbol{a})$ la matrice jacobienne de F, *i.e.* la matrice $p \times n$ suivante

$$J_F(\boldsymbol{a}) = \nabla F(a) = \left(\frac{\partial F_i}{\partial a_j}(\boldsymbol{a}) \right)_{i,j} \quad (6.5)$$

Avec (6.5), le gradient de e est donné par

$$\nabla e(\boldsymbol{a}) = J_F(\boldsymbol{a})^T F(\boldsymbol{a}) \tag{6.6}$$

Dans la méthode de Levenberg-Marquardt, la matrice Hessienne

$$H(\boldsymbol{a}) = \frac{\partial^2 e}{\partial a_i \partial a_j}(\boldsymbol{a})$$

de e est approchée par

$$H(\boldsymbol{a}) \approx J_F(\boldsymbol{a})^T J_F(\boldsymbol{a}), \tag{6.7}$$

beaucoup plus facile à évaluer que la matrice Hessienne exacte.

On considère maintenant un processus de minimisation itératif avec une mise à jour de la forme

$$\boldsymbol{a}^{k+1} = \boldsymbol{a}^k + \boldsymbol{d}^k, \quad k = 0, 1, \ldots,$$

où $\boldsymbol{d}^k$ est une direction de descente de e en $\boldsymbol{a}^k$, *i.e.* une direction qui vérifie

$$\nabla e(\boldsymbol{a}^k)^T \boldsymbol{d}^k < 0.$$

Si on utilise l'algorithme de la plus *forte pente* (voir par exemple [3, 2, 16]) on a

$$d^k = -\lambda_k \nabla e(\boldsymbol{a}^k), \tag{6.8}$$

où $\lambda_k > 0$ est le pas de déplacement. Dans le cas où la matrice Hessienne (6.7) est définie positive (ce qui n'est pas toujours le cas!!), on peut prendre $\boldsymbol{d}^k$ comme solution du système linéaire

$$H(\boldsymbol{a}^k) d^k = -\nabla e(\boldsymbol{a}^k). \tag{6.9}$$

L'idée principale consiste à utiliser (6.8) lorsqu'on est loin de l'optimum, puis de lui substituer (6.9) à l'approche de l'optimum. On remplace alors (6.8) et (6.9) par une formule unique

$$\big(H(\boldsymbol{a}^k) + \lambda_k I_n\big) d^k = -\nabla e(\boldsymbol{a}^k), \tag{6.10}$$

où I_n est la matrice identité d'ordre n. Le paramètre λ_k doit être ajuster pour que $\boldsymbol{d}^k$ solution de (6.10) soit toujours une direction de descente. L'ajustement consiste à augmenter λ_k si le déplacement augmente la fonction (*i.e.* $e(\boldsymbol{a}^k + \boldsymbol{d}^k) > e(\boldsymbol{a}^k)$) et à la diminuer si la fonction diminue dans la direction $\boldsymbol{d}^k$.

Algorithme de Levenberg-Marquardt

Initialisation $k = 0$, $\boldsymbol{a}^0$ donnés, $\lambda_0 = 0.001$

Répéter $k \geq 0$

Calculer $\boldsymbol{d}^k$ solution de

$\left(J_F(\boldsymbol{a}^k)^T J_F(\boldsymbol{a}^k) + \lambda_k \mathbb{I}_n\right) \boldsymbol{d}^k = -J_F(\boldsymbol{a}^k)^T F(\boldsymbol{a}^k)$

Si $e(\boldsymbol{a}^k + \boldsymbol{d}^k) < e(\boldsymbol{a}^k)$ (descente)

$\lambda^{k+1} = \lambda^k/10$

$\boldsymbol{a}^{k+1} = \boldsymbol{a}^k + \boldsymbol{d}^k$

$k = k + 1$

Sinon (pas de descente)

$\lambda_{k+1} = 10\lambda_k$

$\boldsymbol{a}^{k+1} = \boldsymbol{a}^k$

$k = k + 1$

fin si

Jusqu'à $\| J_F(\boldsymbol{a}^k)^T F(\boldsymbol{a}^k) \|_2 < \varepsilon$

Pour le test d'arrêt, on prendra $0.0001 \leq \varepsilon \leq 0.01$. Le fait d'augmenter λ quand $e(\boldsymbol{a}^k + \boldsymbol{d}^k) \geq e(\boldsymbol{a}^k)$ signifie qu'on renforce la diagonale de la matrice pour s'éloigner de la matrice singulière.

Programmer la méthode de Levenberg-Marquardt pour les problèmes de moindres carrés non linéaires. La mise au point du programme se fera sur l'échantillon bidimensionnel suivant

```
x 0.01  1.93  2.95  3.26  4.18  5.73  6.29  7.70  8.91  9.12
y 0.98  0.84  0.80  0.78  0.82  0.78  0.80  0.85  0.90  0.95
```

On fait l'hypothèse que le modèle non linéaire à ajuster est une somme de M gaussiennes, i.e.

$$y(x; \boldsymbol{a}) = \sum_{l=1}^{M} a_{3l-2} \exp\left[-\left(\frac{x - a_{3l-1}}{a_{3l}}\right)^2\right].$$

Les paramètres à trouver sont donc a_{3l-2}, a_{3l-1} et a_{3l} pour $l = 1, \ldots, M$. Ajuster le modèle pour différentes valeurs de $M \leq 5$. Tracer, pour chaque valeur de M, l'échantillon et la courbe d'ajustement.

Index

Symboles et mots clés

' ... 22, 24
* ... 28, 29
+ ... 27
, ... 17, 21
- ... 27
.* ... 31
... ... 17
./ ... 31
/ ... 29
: ... 22, 32
; ... 16, 17, 21, 23
[] ... 23, 24
% ... 17
abs ... 95
all ... 107
amd ... 82, 84
angle ... 95
ans ... 16
any ... 107
axis ... 62
break ... 106
chol ... 52, 81
clabel ... 72
clear ... 19, 120
colperm ... 93
continue ... 106
contour ... 72
delaunay ... 70
delsq ... 82
det ... 47
diag ... 35
diary ... 20
dot ... 29
eig ... 49
error ... 106
eye ... 25
fclose ... 116
feval ... 101
figure ... 59
find ... 79, 108
fopen ... 116
format ... 16
fprintf ... 116
fscanf ... 116
full ... 77
function ... 97
global ... 99
gmres ... 88, 91
griddata ... 69
gtext ... 63
help ... 13, 14, 98
hilb ... 26
hold ... 60
ichol ... 87, 89
ilu ... 85, 91
imag ... 95
intersect ... 114
inv ... 47
ischar ... 96
isfinite ... 108
isinf ... 108
iskeyword ... 18
islogical ... 96
ismember ... 114
isnan ... 107
isreal ... 96
i ... 16
j ... 16
legend ... 63
length ... 23
linspace ... 22, 69

load ... 20, 115
logspace ... 22
lookfor ... 15, 98
lu ... 50, 80
max ... 110
meshgrid ... 66
mesh ... 68
min ... 110
nargin ... 99
nargout ... 99
nnz ... 78
normest ... 38
norm ... 37
null ... 47
numgrid ... 82
ones ... 26
pascal ... 24
pause ... 106
pcg ... 88
pinv ... 48
plot3 ... 66
plot ... 14, 59
print ... 64
prod ... 112
qr ... 54
randn ... 26
rand ... 26
real ... 95
reshape ... 32
return ... 98, 106
save ... 20, 116
setdiff ... 114
setxor ... 114
size ... 23, 24
sort ... 109
sparse ... 77
spdiags ... 78, 93
speye ... 78
spfun ... 78
sprandn ... 78
sprandsym ... 78
sprand ... 78
spy ... 79
subplot ... 64
sum ... 112
surf ... 68
svd ... 55
symamd ... 82, 84
symrcm ... 82
text ... 63
tic ... 79
title ... 63
toc ... 79
triplot ... 70
union ... 113
view ... 68
whos ... 19
who ... 19
xlabel ... 62
ylabel ... 62
zeros ... 120
zeros ... 25
.^ ... 31
\ ... 30, 43

A
aide en ligne ... 13

C
commentaire ... 17
condition d'une matrice ... 84
constantes prédéfinies ... 18
contrôle du pivotage ... 80
conversion matrice/vecteur ... 32
courbe
  2D ... 59
  3D ... 66
  axe ... 62
  de niveau ... 72

D
déterminant ... 47
différences finies ... 82

F
factorisation
  de Cholesky ... 52
  incomplète ... 85
  LU ... 50
  QR ... 54
fichier
  écriture ... 117
  fermeture ... 116
  lecture ... 117
  ouverture ... 116
figure
  insérer du texte ... 63
  légende ... 63
  sauvegarde ... 64

titre 63
fonction
logique 107

I
intersection de deux matrices 114

K
Krylov (sous-espace) 90

M
méthode
du gradient conjugué préconditionné 88
GMRES 90
Newton-Raphson 101
matrice
aléatoire 26
d'indices 34
de Hilbert 26
de Pascal 26
de Vandermonde 41
diagonale 35
identité 25
inverse 47
ne contenant que des 1 26
nulle 25
orthogonale 54
par blocs 37
pseudo-inverse 45, 48
suppression d'une colonne 35
suppression d'une ligne 35
taille 24
transposée 24
unitaire 54
matrice creuse
conversion 77
création 77
du Laplacien 82
nombre d'éléments non nuls 78
stockage 77
visualiser 79
maximum 110
méthode
Levenberg-Marquardt 122
Newton-Raphson 121
minimum 110
moindres carrés
non linéaires 122
mots réservés 18
mots-clés 18

N
norme
de Frobenius 39
matricielle 38
matricielle subordonnée 38
vectorielle 37

O
opérateur
de comparaison 104
opérateurs
logiques 105
opération
élément par élément 31
addition 27
division 29
produit de vecteurs 28
produit matriciel 29
produit scalaire 28, 29
puissance 30
soustraction 27

P
préconditionnement d'un système linéaire 84
problème
de moindres carrés 45
produit 112

R
régression linéaire 45
réunion de deux matrices 113

S
somme 112
surface
analytique 66
définie par un ensemble de points 69
système
aux équations normales 45
carré 43
sous-déterminé 46
surdéterminé 45

T
tabuler une fonction 31
tri 109
triangulation 70
type
chaîne 95
complexe 95

logique 96

V

valeur
propre 49
singulière 55
variables prédéfinies 18
vecteur
colonne 21, 23
d'indices 33
ligne 21, 23
propre 49
taille 23
transposé 22
vectorisation 118

Bibliographie

[1] AMESTOY P.R., DAVIS T.A. and DUFF I.S. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Anal. Appl.*, 17(4):886–905, 1996.

[2] BONNANS J.-F., GILBERT J.C., LEMARÉCHAL C. and SAGASTIZABAL C. *Optimisation Numérique - Aspects théoriques et pratiques*, volume 27 of *Mathématiques et Applications*. Springer-Verlag, 1997.

[3] BONNANS J.-F., GILBERT J.C., LEMARÉCHAL C. and SAGASTIZABAL C. *Numerical Optimization: theoretical and numerical aspects*. Springer-Verlag, 2003.

[4] DAVIS T.A. *Direct Methods for Sparse Matrices*. SIAM, Philadelphia, 2006.

[5] DELANNOY C. *Programmer en Fortran 90*. Eyrolles, Paris, 1993.

[6] DENNIS J.E. and SCHNABEL R.B. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Classics in Applied Mathematics. SIAM, Philadelphia, 1996.

[7] GOLUB G.H. and ORTEGA J.M. *Scientific Computing and Differential Equations*. Academic Press, 1992.

[8] GOLUB G.H. and VAN LOAN C.F. *Matrix Computations*. The John Hopkins University Press, Baltimore, 1989.

[9] GOOSSENS M., MITTELBACH and SAMARIN A. *LaTeX Companion*. CampusPress, Paris, 2001.

[10] KOPKA H. and DALY P.W. *Guide to LaTeX*. Addison-Wesley, 2004.

[11] LASCAUX P. and THÉODOR R. *Analyse numérique matricielle appliquée à l'art de l'ingénieur*, volume 1. Masson, Paris, 1994.

[12] LASCAUX P. and THÉODOR R. *Analyse numérique matricielle appliquée à l'art de l'ingénieur*, volume 2. Masson, Paris, 1994.

[13] LUCQUIN B. *Equations aux Dérivvées Partielles et leurs approximations*. Ellipses, Paris, 2004.

[14] LUENBERGER D. *Linear and Nonlinear Programming*. Addison Wesley, Reading, MA, 1989.

[15] MEURANT G. *Computer Solution of Large Systems*. Studies in Mathematics and its Applications. North Holland, 1999.

[16] MINOUX M. *Programmation Mathématique: Théorie et Algorithmes*. Tec & Doc Lavoisier, Paris, 2008.

[17] ORTEGA J.M. and RHAINBOLDT W.C. *Iterative Solution of Nonlinear Equations in Several Variables*. Academic Press, New York, 1970.

[18] PRESS W. H., TEUKOLSKY S. A., VETTERLING W. T. and FLANNERY B.P. *Numerical Recipes*. Cambridge University Press, 1992.

[19] QUARTERONI A., SACCO R. and SALERI F. *Méthodes numériques pour le calcul scientifique*. Springer, 2000.

[20] SIBONY M. and MARDON J.-C. *Approximation et équations différentielles*. Hermann, Paris, 1988.