Introduction à Matlab et Octave

B. Torrésani
Université de Provence

Master Mathématiques et Applications

Spécialité GSI

Année Universitaire 2009-10, premier semestre

1 Introduction

Quelques lignes d'introduction[1].

Matlab est un logiciel (commercial) de calcul numérique, de visualisation et de programmation très performant et convivial[2]. Le nom de Matlab vient de MATrix LABoratory, les éléments de données de base manipulés par Matlab étant des matrices (pouvant bien évidemment se réduire à des vecteurs et des scalaires) qui ne nécessitent ni dimensionnement ni déclaration de type. Contrairement aux langages de programmation classiques (scalaires et à compiler), les opérateurs et fonctions Matlab permettent de manipuler directement et interactivement ces données matricielles, rendant ainsi Matlab particulièrement efficace en calcul numérique, analyse et visualisation de données en particulier.
Mais Matlab est aussi un environnement de développement ("progiciel") à part entière : son langage d'assez haut niveau, doté notamment de structures de contrôles, fonctions d'entrée-sortie et de visualisation 2D et 3D, éditeur/debugger, outils de construction d'interface utilisateur graphique (GUI)... permet à l'utilisateur d'élaborer ses propres fonctions ainsi que de véritables programmes ("M-files") appelés scripts vu le caractère interprété de ce langage.
Matlab est disponible sur tous les systèmes d'exploitation standards (Windows, Unix/Linux, MacOS X...) et son architecture est relativement ouverte. Le champ d'utilisation de Matlab peut être étendu aux systèmes non linéaires et aux problèmes associés de simulation avec le produit complémentaire SIMULINK. Les capacités de Matlab peuvent en outre être enrichies par des fonctions spécialisées regroupées au sein de dizaines de "toolboxes" (boîtes à outils qui sont des collections de "M-files") couvrant des domaines très variés tels que :
- Traitement de signaux (et du son en particulier)
- Traitement d'image, cartographie
- Analyse de données
- Statistiques
- Finance et mathématiques financières
- Mathématiques symboliques (accès au noyau Maple V)
- Analyse numérique (accès aux routines NAG)

Une interface de programmation applicative (API) rend finalement possible l'interaction entre Matlab et les environnements de développement classiques (exécution de routines C ou Fortran depuis Matlab , ou accès aux fonctions Matlab depuis des programmes C ou Fortran). Matlab permet en outre de déployer de véritables applications à l'aide des outils de conversion optionnels suivants :
- Matlab → code C/C++, avec le Matlab Compiler
- Matlab → Excel add-ins, avec le Matlab Excel Builder
- Matlab → objets COM Windows, avec le Matlab COM Builder

Toutes ces caractéristiques font aujourd'hui de Matlab un standard incontournable en milieu académique, dans les différents domaines de l'ingénieur et la recherche scientifique.
GNU Octave , et autres alternatives à Matlab : Matlab est un logiciel commercial qui coûte relativement cher, même au tarif académique. Cependant, il existe des logiciels libres et open-source analogues voire compatibles avec Matlab , donc gratuits, également multi-plateformes :
- GNU Octave : logiciel offrant la meilleure compatibilité par rapport à Matlab (qualifiable de "clone Matlab ", surtout depuis la version Octave 2.9/3.0 et avec les packages du repository Octave -Forge).
- FreeMat : logiciel libre compatible avec Matlab et Octave , déjà très abouti, avec IDE comprenant : editor/debugger, history, workspace tool, path tool, file browser, 2D/3D graphics...
- Scilab : logiciel libre "analogue" à Matlab et Octave en terme de fonctionnalités, très abouti, plus jeune que Octave mais beaucoup moins compatible avec Matlab (fonctions et syntaxe différentes...)

[1] adaptées du site web du cours Matlab /Octave de l'EPFL : `http ://enacit1.epfl.ch/cours_matlab`
[2] Développé par la société *The MathWorks*

et ne permettant donc pas une réutilisation aisée de scripts.
– R : un logiciel libre, davantage orienté vers les statistiques.
– NumPy : Numerical Python, une extension de Python tournée vers le calcul scientifique
– ...

Dans ce cours, on se focalisera sur l'utilisation de Matlab et Octave , qui sont en très grande partie compatibles. Les TP étant effectués avec Matlab , les instructions données sont des instruction Matlab ; ceci dit, elles sont également opérationnelles sous Octave .
Octave est téléchargeable librement sur le site

`http ://www.gnu.org/software/octave/`

1.1 Prise en main

1.1.1 Matlab/Octave pour les nuls

En utilisant la commande `mkdir`, commencer par créer un répertoire, par exemple `MonMatlab` (ou autre), dans lequel vous stockerez votre travail. Placez vous dans ce répertoire (en utilisant la commande `cd`). Téléchargez dans ce répertoire les fichiers se trouvant sur

`http ://www.cmi.univ-mrs.fr/~torresan/Matlab10/Fonctions/moyennevariance.m`
`http ://www.cmi.univ-mrs.fr/~torresan/Matlab10/Data/gspi35_2.wav`

En ligne de commande (dans une fenêtre "terminal"), taper `matlab &` (rappel : le symbole "&" signifie que le logiciel est lancé en tâche de fond). Normalement, l'interface de matlab s'affiche. Examiner les différentes composantes de celle-ci : fenêtre de commande, espace de travail, répertoire courant, historique des commandes,...

Exercice 1.1 Dans la fenêtre de commande, taper

```
>> a = 1+1
```

puis

```
>> b = 1+2 ;
```

La première ligne calcule 1+1, l'affecte dans la variable `a`, et affiche le résultat. La seconde calcule 1+2, l'affecte dans la variable `b`, et n'affiche pas le résultat à cause de la présence du " ;" en fin de ligne. Pour afficher de nouveau le résultat, taper par exemple

```
>> a
```

Noter que les variables ainsi créées sont également visibles dans l'onglet "Workspace". Noter aussi que lorsque le résultat du calcul n'est pas affecté à une variable, il est affecté à une variable par défaut notée `ans`.

La commande qui sauve : `help`, accessible soit dans la fenêtre de commande

```
>> help doc
```

soit sous format `HTML`, grâce à la commande `doc`, soit directement à partir du menu de l'interface Matlab (en haut à droite). Il est aussi possible de rechercher une fonction à partir d'un mot clé, grâce à la commande `lookfor`[3]. Par exemple

```
>> lookfor('variance')
```

Exercice 1.2 Se documenter sur les variables `who`, `whos`, et `clear`, et les essayer.

Quitter Matlab :

```
>> exit
```

1.1.2 Gestion des répertoires, fichiers,...

Relancer Matlab depuis votre répertoire de travail. Dans la fenêtre de commande, tester les instructions `pwd` et `ls`. Les informations obtenues via ces commandes sont également accessibles dans l'onglet "Current Directory".

[3]Alternative : utiliser le moteur de recherche de Matlab disponible dans le help.

MATLAB permet de sauvegarder des variables dans des fichiers de données, dont le nom porte l'extension "`.mat`", et de charger de tels fichiers. Se documenter sur les instructions `save` et `load`. Pour vérifier, interpréter puis effectuer les commandes suivantes :

```
>> A = 1 :10 ;
>> B = 3 ;
>> who
>> save Tout.mat
>> ls
>> save Aseul.mat A
>> clear all
>> load Aseul.mat
>> who
>> load Tout.mat
>> ls
```

Ceci vous permettra de sauvegarder des résultats d'une séance à l'autre.
Il existe également des fonctions de lecture/écriture de plus bas niveau, plus proche du langage `C` (`fopen, fclose, fprintf, fscanf, fwrite,...`). Il est également possible de lire (et écrire) des fichiers plus spécifiques, par exemple des fichiers son aux formats `.wav` ou `.au` (fonctions `wavread, auread`, ou des fichiers image au format `.jpg` (fonction `readjpg`,...). On y reviendra plus tard.

1.2 Vecteurs, matrices,...

L'élément de base sous MATLAB est la matrice (le vecteur, ligne ou colonne, étant considéré comme une matrice particulière). On peut accéder aux dimensions des matrices en utilisant l'instruction `size`, par exemple

```
>> A = rand(2,3) ;
>> tmp = size(A) ;
>> tmp(1)
>> tmp(2)
```

Les règles de multiplication sont par défaut les règles de multiplication matricielle.

1.2.1 Manipulations de base

Pour générer "manuellement" une matrice, il faut savoir que le séparateur de colonnes est la virgule (ou l'espace), et le séparateur de lignes est le "point virgule". Pour vérifier :

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

Dans les deux expressions suivantes

```
>> x = 1 :100
>> x = 1 :10 :100
```

on génère deux vecteurs (ligne) constitués des 100 premiers entiers positifs dans le premier cas, et des entiers de 1 à 100 par pas de 10 dans le second cas.

L'opération " ' " représente la conjugaison Hermitienne des matrices (la transposition suivie de la conjugaison complexe). La transposition des matrices s'obtient en associant la conjugaison Hermitienne à la conjugaison complexe. Pour vérifier :

```
>> A=[1,i ;2i,2 ;3i,3]
>> AH = A'
>> AT = conj(A')
```

EXERCICE 1.3 Après en avoir vérifié la syntaxe à l'aide du `help`, tester les fonctions `ones`, `zeros`, `eye`, `rand` et `randn`.

EXERCICE 1.4
1. Générer deux matrices "multipliables" `A` et `B`, et les multiplier.
2. Générer deux matrices "non-multipliables" `A` et `B`, et essayer de les multiplier.

Matlab implémente également un autre type d'opération sur les matrices (et les vecteurs) : les opérations "point par point", dites "pointées". Ces opérations ne sont possibles qu'entre matrices de même taille. Tester cette opération, par exemple

```
>> A = [1,2,3 ;4,5,6] ;
>> B = [4,5,6 ;7,8,9] ;
>> A.*B
>> A./B
```

ou encore

```
>> x = 1 :5
>> x.^ 2
```

1.2.2 Algèbre linéaire

Matlab a été originellement créé pour l'algèbre linéaire (matlab signifie « matrix laboratory »). Ainsi, l'objet de base est la matrice, et de nombreuses opérations matricielles sont déjà implémentées et optimisées.

Exercice 1.5

1. Se documenter sur les instructions `inv` et `det`. Générer deux matrices carrées de même taille `A` et `B`, vérifier qu'elles sont inversibles. Pour vérifier :

```
>> A/B
>> A*inv(B)
>> A\B
>> inv(A)*B
```

2. Générer deux vecteurs de même taille (ligne ou colonne), et effectuer leur produit scalaire (ou leur produit Hermitien s'ils sont complexes). On pourra procéder de deux façons :
   – en utilisant l'instruction `sum` et une multiplication pointée
   – en utilisant une transposition et un produit matriciel
   Dans les deux cas, ça ne prend qu'une ligne.
3. Utiliser les opérations matricielles pour résoudre le système linéaire suivant (après avoir testé l'existence de solutions) :

$$\left\{\begin{array}{ccccccc} x & + & 2y & + & 3z & = & 1 \\ 4x & + & 5y & + & 6z & = & 2 \\ 7x & + & 8y & + & 10z & = & 3 \end{array}\right.$$

4. Se documenter sur les commande `null` et `eig`. Utiliser ces deux commandes pour résoudre le système ci-dessus en utilisant la diagonalisation des matrices.

1.2.3 Fonctions de vecteurs et de matrices

La plupart des fonctions usuelles (trigonométriques, exponentielles,...) existent également sous forme vectorielle et matricielle. Par exemple, tester la suite d'expressions

```
>> xxx = linspace(1,10,100) ;
>> C = cos(2*pi*xxx) ;
```

On pourra utiliser l'instruction

```
>> plot(C)
```

pour visualiser le résultat.

De même, pour les matrices, l'expression `exp(A)` renvoit la matrice dont les éléments sont les exponentielles des éléments de matrice de `A` : c'est une opération pointée,

```
>> A = eye(2)
>> B = exp(A)
```

à ne pas confondre avec l'exponentielle matricielle des matrices carrées

$$e^A = \sum_{k=0}^{\infty} \frac{A^k}{k!}$$

pour laquelle est définie la fonction `expm` (se documenter)

```
>> B = expm(A)
```

1.2.4 Au delà des matrices

Matlab permet également de manipuler des tableaux à plus d'une ou deux entrées, comme par exemple des tableaux “cubiques”, ou plus. Par exemple

```
>> A = randn(2,2,2)
```

1.3 Graphiques élémentaires

Matlab possède d'intéressantes possibilités graphiques. Il est possible d'ouvrir une (nouvelle) fenêtre graphique grâce à l'instruction `figure`. Les fenêtres sont numérotées. On se place dans une fenêtre donnée (par exemple la quatrième) en utilisant de nouveau l'instruction `figure` (dans ce cas, en effectuant `figure(4)`. Il est également possible de tracer plusieurs graphiques dans la même figure grâce à l'instruction `subplot` (se documenter).

1.3.1 Tracer une suite de valeurs, ou de points

La fonction de base est la fonction `plot`. Par exemple, on pourra (après les avoir interprêtées) essayer les instructions

```
>> x = linspace(0,10,100) ;
>> f = exp(-x/10).* cos(2*pi*x) ;
>> g = f + randn(size(x))/3 ;
>> plot(x,g)
>> hold on
>> plot(x,f,'r')
>> xlabel('Abscisses')
>> ylabel('Ordonnées')
>> title('Ceci est un titre')
```

On a utilisé ici l'instruction `hold on`, qui permet de conserver un tracé et en superposer un autre (par défaut, un nouveau tracé efface le précédent, ce qui correspond à l'instruction `hold off`), et l'argument `'r'` de la fonction `plot`, qui permet de tracer en rouge au lieu du bleu (couleur par défaut). D'autres couleurs sont possibles, se documenter. Se documenter sur les fonctions `xlabel` et `ylabel`. Le résultat de ces quelques lignes se trouve en Fig. 1.1 (figure de gauche).
Il existe également de multiples fonctions graphiques avec lesquelles l'on peut jouer pour améliorer l'aspect du graphe généré. Par exemple, en remplaçant les trois dernières lignes ci-dessus par

```
>> xlabel('Abscisses'),'fontsize',20)
>> ylabel('Ordonnées','fontsize',20)
>> title('Ceci est un titre','fontsize',24)
```

on modifie les tailles des caractères dans les légendes et le titre (voir tracé de droite). Il est aussi possible d'effectuer ces modifications directement sur la fenêtre graphique, en l'éditant.

1.3.2 Tracer des lignes et des polygones

L'instruction `line` permet de tracer une ligne dans la fenêtre graphique. La syntaxe est la suivante : `line([x1,x2,...xn],[y1,y2,...yn])` trace un polygone à n côtés, dont les sommets ont pour coordonnées `(x1,y1)`,... Par exemple, les lignes

```
>> x = [1,1.5,2,1.5,1] ;
>> y = [1,0,1,0.5,1] ;
>> line(x,y)
```

génèrent un “chevron” passant par les points de coordonnées (1,1), (1.5,0), (2,1), (1.5,0.5) et (1,1) (voir Figure 1.2

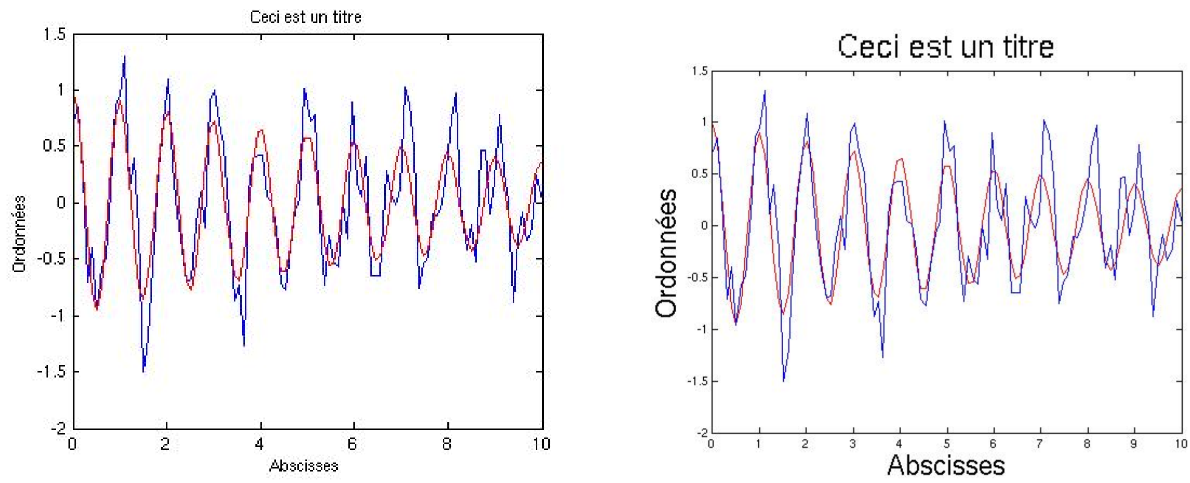

FIG. 1.1 – Exemple de représentation graphique

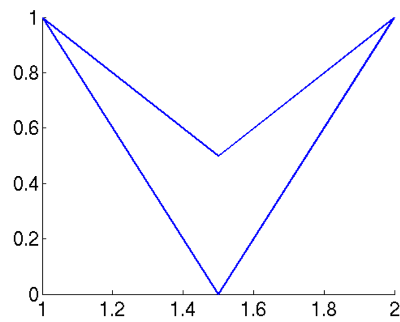

FIG. 1.2 – Un “chevron” tracé avec l’instruction `line`.

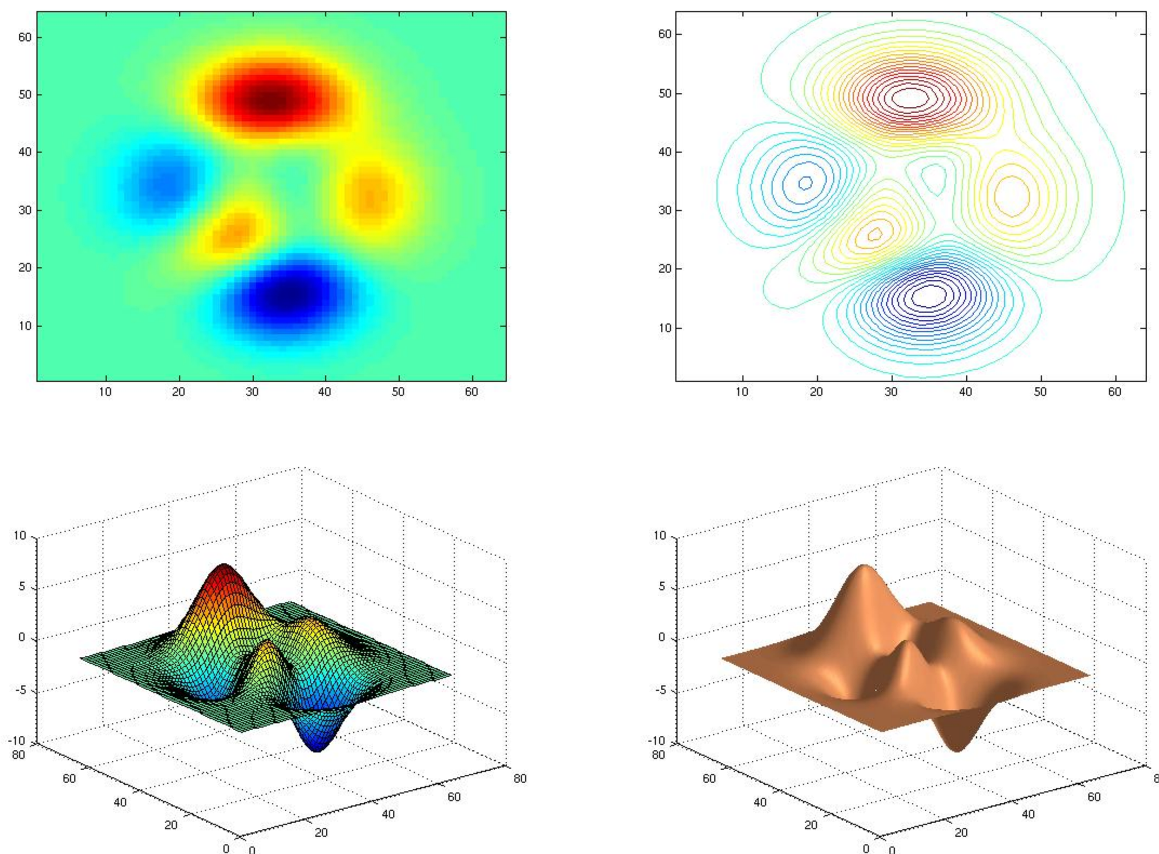

FIG. 1.3 – Différentes visualisations graphiques d'une fonction de deux variables

EXERCICE 1.6 Pour vérifier, tracer un losange centré sur un point donné.

1.3.3 Graphiques plus évolués

- Les instructions `image`, `imagesc` permettent de représenter une matrice sous forme d'une image. Se documenter, et imager une matrice aléatoire
- Il existe des instructions permettant de faire du graphique tridimensionnel (`plot3, mesh, surf,...`), on les verra plus tard.

Les quelques lignes suivantes permettent de donner différentes représentations graphiques d'une fonction de 2 variables (la fonction `peaks`, un classique de MATLAB)

```
>> Z = peaks(64) ;
>> imagesc(Z) ; axis xy (représentation sous forme d'image)
>> contour(Z,32) ; (32 lignes de niveau)
>> surf(Z) ;
>> surfl(Z) ; shading interp ; colormap('copper') ;
```

Les résultats se troucvent dans la FIG. 1.3 (où on a également joué avec la `colormap` pour la dernière image).

1.3.4 Utilisation des fonctionnalités de la fenêtre graphique

Une fois un graphique tracé, il est toujours possible de le modifier en utilisant les outils graphiques de MATLAB . On peut par exemple modifier les couleurs des courbes, les fontes des labels, titres et légendes et leur taille,...

Il est également possible de sauvegarder un graphique sous divers formats. Le meilleur est a priori le format "`.fig`", qui permet d'éditer le graphique lors de sessions futures. Il est également possible de sauvegarder

aux formats ps (postscript), eps (postscript encapsulé, c'est à dire sans information de page), jpg (format JPEG, compression avec perte),...

1.4 Scripts et fonctions

Un script est un fichier texte, contenant une série d'instructions MATLAB . Ces instructions sont exécutées lorsque le script est exécuté. Les variables crées lors de l'exécution du script restent en mémoire après exécution. L'extension 'standard' d'un fichier de script est '.m'.

Une fonction diffère d'un script par l'existence de variables d'entrée et de sortie. Les variables introduites dans la fonction, autres que les variables de sortie, sont effacées après exécution de la fonction.
La syntaxe est la suivante :

```
function [variables_sortie] = mafonction(variables_entrée)
```

Cette fonction sera sauvée dans le fichier "`mafonction.m`".

Par exemple, la fonction ci-dessous[4] prend comme variable d'entrée un vecteur de longueur quelconque, et retourne sa moyenne et sa variance

```
function [m,v] = moyennevariance(X)
% function : moyennevariance
%  usage : [m,v] = moyennevariance(X)
%  Variable d'entrée :
%    X : vecteur
%  Variables de sortie :
%    m : moyenne (scalaire)
%    v : variance (scalaire)
longueur = length(X) ;
m = sum(X)/longueur ;
v = sum((X-m).^2)/(longueur-1) ;
```

Dans cette fonction, les lignes précédées d'un signe % sont des lignes de commentaires. L'intérêt d'insérer de telles lignes dans une fonction est double : d'une part, elles permettent de se souvenir des significations des variables, et d'autre part, ce commentaire apparaît lorsque l'on effectue

```
>> help(moyennevariance)
```

L'interface utilisateur MATLAB contient un éditeur de texte permettant d'écrire des scripts et des fonctions.

EXERCICE 1.7 Ecrire une fonction de visualisation de la transformée de Fourier discrète (TFD)

$$\hat{X}_k = \sum_{n=0}^{N-1} X_n e^{-2i\pi kn/N}$$

d'un vecteur de taille fixée N. On utilisera pour cela la fonction `fft`, qui calcule la TFD. La fonction aura pour argument d'entrée un vecteur, calculera sa transformée de Fourier, et tracera le module (fonction `abs`) de sa transformée de Fourier. Elle retournera la transformée de Fourier.

[4] Inutile car MATLAB possède déjà une fonction `mean` et une fonction `var`.

1.5 Devoir

Les exercices suivants sont à faire individuellement, et à rendre sous forme d'un fichier "archive", à envoyer à l'adresse `torresan@cmi.univ-mrs.fr`. Pour cela, sous unix/linux, créer un répertoire nommé `nom_prenom_tp1`, et copier tous les fichiers `*.*` dans ce répertoire. Ensuite, dans le répertoire supérieur, effectuer `tar -zcvf nom_prenom_tp1.tgz nom_prenom_tp1/*`. Sous Windows, on pourra utiliser un logiciel de compactage (winzip, winrar,7zip,...) pour créer l'archive.

EXERCICE 1.8 Ecrire une fonction `HistConv.m`, prenant pour argument un entier positif `N`, qui génère `N` nombres pseudo-aléatoires uniformément distribués, trace l'histogramme correspondant et affiche la moyenne et la variance de la population ainsi créée. On pourra utiliser la fonction `lookfor` pour trouver les fonctions MATLAB nécessaires.

EXERCICE 1.9 Ecrire une fonction `calculecorr.m` qui prend comme variables d'entrée deux vecteurs de même dimension, calcule le coefficient de corrélation entre ces vecteurs et l'affiche dans la fenêtre de commande. On rappelle que le coefficient de corrélation est donné par

$$r = \frac{1}{N-1} \frac{\sum_{n=1}^{N}(X_n - \overline{X})(Y_n - \overline{Y})}{\sigma_X \sigma_Y} ,$$

où $\overline{X}$ est la moyenne de X, σ_X est l'écart-type de X, et de même pour Y. On pourra tester cette fonction en utilisant des vecteurs aléatoires (`randn`).

EXERCICE 1.10 Ecrire une fonction `resolsystlin.m`, prenant en entrée un vecteur à 4 composantes $y = (y_1, y_2, y_3, y_4)$, et retournant en sortie la solution de l'équation

$$\left\{ \begin{array}{ccccccccc} x_1 & + & 2x_2 & + & 3x_3 & + & x_4 & = & y_1 \\ 4x_1 & + & 5x_2 & + & 6x_3 & + & x_4 & = & y_2 \\ 7x_1 & + & 8x_2 & + & 10x_3 & + & x_4 & = & y_3 \\ 9x_1 & + & 11x_2 & + & 12x_3 & + & x_4 & = & y_4 \end{array} \right.$$

EXERCICE 1.11 Ecrire une fonction `tracepolygone.m` prenant en entrée un entier positif `N`, et dessinant dans la fenêtre graphique un polygone régulier à `N` côtés.

EXERCICE 1.12 Ecrire une fonction `proj2.m` prenant en entrée une matrice A réelle symétrique $N \times N$ et un vecteur V à N lignes, et effectuant les opérations suivantes :
- Diagonalisation de A, en utilisant la fonction `eig`.
- Sélection des deux valeurs propres les plus grandes, en utilisant la fonction de tri `sort`
- Projection du vecteur V sur le plan engendré par les deux vecteurs propres correspondants.

Les variables de sortie seront la projection de V sur le plan, ainsi que les composantes de V sur les deux vecteurs propres.

2 Eléments de programmation

Pour commencer, deux définitions :
- **Algorithme :** ensemble de règles précises, qui permettent d'obtenir un résultat à partir d'opérations élémentaires.
- **Algorithme numérique :** Les données et les résultats sont des nombres.

Cela va sans dire, mais mieux encore en le disant : un programme ne se conçoit pas sans une analyse préalable du problème posé, c'est à dire une analyse des points suivants :
- Quel est le problème posé ? quel est le résultat attendu ?
- Quelles sont les données de départ ?
- Quelles sont les données que doit retourner le programme ? les représentations graphiques ?
- Quel est l'algorithme de calcul ?
- Comment "optimiser" le calcul ? Comment le décomposer en étapes indépendantes ?

Une fois cette analyse faite et écrite, la phase de codage proprement dite peut commencer.

2.1 Quelques éléments de programmation sous Matlab et Octave

Un programme Matlab ne diffère pas fondamentalement d'un programme dans un langage classique. La seule différence importante tient dans le fait que la majorité des opérations par défaut dans Matlab étant vectorisées, il est généralement préférable d'exploiter cette caractéristique, par exemple en évitant des boucles dans les programmes.

2.1.1 Connecteurs logiques et de relation

Les connecteurs de relation (qui retournent soit 0 soit 1 suivant que l'expression correspondante est vraie ou fausse), sont résumés dans le tableau suivant.

Connecteur	Signification
$<$	plus petit
$>$	plus grand
$<=$	plus petit ou égal
$>=$	plus grand ou égal
$==$	égal
$\sim=$	différent

Par exemple, l'instruction

```
>> a == b
```

retourne 1 si le contenu de la variable `a` est égal au contenu de la variable `b`, et 0 sinon.

Exercice 2.1 Ecrire une fonction `monmax.m`, prenant en entrée deux nombres, et retournant le plus grand des deux (sans utiliser la fonction `max`,... sinon c'est de la triche).

Exercice 2.2 Interpréter puis tester (ou l'inverse...) les deux lignes suivantes

```
>> x = randn(10,1) ;
>> y = (abs(x)>=1)
```

On essaiera aussi la fonction `find` :

```
>> z = find(abs(x)>=1)
```

Quant aux connecteurs logiques (opérant en binaire), les plus importants se trouvent dans le tableau suivant

Connecteur	Signification
&	et
|	ou
`xor`	ou exclusif
~	non

EXERCICE 2.3 tester ces instructions, après en avoir vérifié la syntaxe à l'aide du `help`.

EXERCICE 2.4 Construire les tables des trois connecteurs &, | et `xor`.

EXERCICE 2.5 Interpréter les commandes :

```
>> x=3 ;
>> y=2 ;
>> (x==3)&(y==2)
>> (x==3)|(y==0)
```

Dans chaque cas, quel sera le résultat ?

2.1.2 Boucles

Les boucles sont l'un des moyens de contrôle de l'exécution du programme. La syntaxe est identique à ce qu'elle est dans la plupart des langages.

– Boucles `for` : on peut créer une boucle en utilisant `for ... end`.

```
>> k2 = zeros(1,5) ;
>> for k = 1 :5
>>   k2(k) = k^2 ;
>> end
```

Il est important de signaler que cette boucle est bien moins efficace que la simple ligne

```
>> k2 = (1 :5).^2
```

On peut aussi réaliser des boucles `for` imbriquées.

– Boucle `while` : On peut créer une boucle en utilisant `while ... end.`

```
>> k = 1 ;
>> k2 = zeros(1,5) ;
>> while tmp <= 25
>>   tmp = k^2 ;
>>   k2(k) = tmp ;
>>   k = k+1 ;
>> end
```

EXERCICE 2.6 Ecrire deux fonctions `fact1.m` et `fact2.m` calculant la factorielle d'un entier, en utilisant une boucle `for` puis une boucle `while`. On pourra utiliser la fonction `factorial` pour vérifier le résultat.

2.1.3 La construction switch-case

La construction `switch-case` permet d'énumérer un certain nombre de cas de figures. Par exemple,

```
>> x = ceil(10*rand) ;
>> switch x
>>   case {1,2}
>>     disp('Probabilite = 20%')
>>   case {3,4,5}
>>     disp('Probabilite = 30%')
>>   otherwise
>>     disp('Probabilite = 50%')
>> end
```

Exercice 2.7 Exécuter et interpréter ce script (qui utilise la fonction `ceil` qui arrondit un réel à l'entier le plus proche, et la fonction `disp`, qui affiche du texte à l'écran).

Remarque : l'exemple précédent permet de voir une utilisation de chaîne de caractères : l'expression `'Probabilité = 20%'` est une chaîne de caractères.

2.1.4 Tests : if, elseif, else

La suite d'instructions `if ... elseif ... else` permet de choisir entre plusieurs options. La syntaxe est là encore similaire à la syntaxe "classique" : par exemple

```
>> x = randn(1) ;
>> if abs(x) <= 1
>>   y = 0 ;
>> elseif abs(x) > 5
>>   y = x^ 2 ;
>> else
>>   y = x ;
>> end
```

Exercice 2.8 Que fait cette suite d'instructions ?

Exercice 2.9 En utilisant la fonction `rand`,
- Générer des points uniformément distribués dans un carré de côté 2 et de centre 0.
- Tester la distance à l'origine du point généré.
- Effectuer ces opérations à l'intérieur d'une boucle, stocker dans un tableau à 2 colonnes les coordonnées des points dont la distance à l'origine est inférieure à 1, et tracer ces derniers.
- Pour compléter, on pourra également tracer les histogrammes des coordonnées Cartésiennes des points ainsi générés, ainsi que de leurs coordonnées polaires.

Remarque : Dans l'exercice précédent, on pourra remarquer que l'on a généré des points pseudo-aléatoires, distribués sur le disque de rayon 1 avec une distribution uniforme, c'est à dire une densité de probabilités de la forme

$$\rho(x,y) = \begin{cases} 1/\pi & \text{si } \sqrt{x^2+y^2} \leq 1 \\ 0 & \text{sinon} \end{cases}$$

En notant R et Θ les variables aléatoires correspondant aux coordonnées radiale et angulaire des points générés, il est facile de calculer leur densité de probabilités, et de les comparer aux résultats obtenus.

2.1.5 Communication avec l'utilisateur... et les fichiers

Matlab offre de nombreux moyens de communication entre un programme et l'utilisateur, par exemple :
- Comme on l'a déjà vu, on peut afficher un message, une valeur à l'écran avec l'instruction `disp` :
  ```
  >> disp('Ceci est un test')
  ```
- On peut entrer une valeur avec l'instruction `input` :
  ```
  >> x = input('Valeur de x = ')
  ```
  Matlab attend alors qu'un nombre soit entré sur le clavier.
- Il est possible d'afficher un message d'erreur, en utilisant `error`.

Il faut remarquer qu'il existe des fonctions plus sophistiquées, telles que `inputdlg`, ou `errordlg`, sur lesquelles on peut se documenter grâce à l'aide. Ces fonctions ouvrent une fenêtre de dialogue.

Matlab permet également d'effectuer des opérations d'entrée-sortie avec des fichiers (et la fenêtre de commande), en s'inspirant de la syntaxe de la programmation en langage `C`. Par exemple, la commande

```
>> x = pi ;
>> fprintf('Le resultat du calcul est x=% f\n',x)
```

affiche la valeur de `x` à la fin de la phrase. Le symbole `%f` est le format dans lequel sera affichée la valeur de `x` (ici, `float`), et le symbole \ n est le “retour charriot”.
Les entrées-sorties avec des fichiers s'effectuent avec une syntaxe similaire, en n'oubliant toutefois pas d'ouvrir le fichier avant de lire ou écrire (instruction `fopen`), et de le refermer ensuite (instruction `close`). `fopen` affecte un identifiant au fichier ouvert, identifiant qu'il faut préciser pour lire ou écrire. Se documenter sur ces instructions, au besoin en utilisant l'exercice qui suit.

EXERCICE 2.10 En vous aidant du `help`, interpréter puis effectuer les lignes de code suivantes :

```
>> fp1 = fopen('toto.dat','w') ;
>> x = pi ;
>> fprintf(fp1,'%f,x) ;
>> fclose(fp1) ;
>> fp2 = fopen('toto.dat','r') ;
>> y = fscanf(fp2,'%f') ;
>> fclose(fp2) ;
>> fprintf('y = %e\n',y) ;
```

EXERCICE 2.11 Compléter votre fonction factorielle en lui faisant vérifier que la variable d'entrée est un entier positif, et la faisant sortir avec un message d'erreur si tel n'est pas le cas. Pour tester que l'argument est un entier, on pourra utiliser la fonction `floor`. Pour tester le signe, utiliser `sign`.

2.2 Marche au hasard 1D

On s'intéresse à un marcheur, se promenant sur l'axe réel, et se déplaçant dans une direction ou l'autre par pas entiers. On suppose qu'à chaque déplacement n, il a une probabilité p d'aller vers la droite ($\Delta_n = 1$), et une probabilité $q = 1 - p$ de se diriger vers la gauche ($\Delta_n = -1$). Sa position à l'instant N est donc

$$X_N = \sum_{n=1}^{N} \Delta_n \ .$$

EXERCICE 2.12
- Ecrire une fonction `hasardbinaire.m`, prenant en entrée une probabilité $p \in [0, 1]$, et retournant en sortie un nombre pseudo-aléatoire valant 1 avec probabilité p et -1 avec probabilité $1 - p$.
- Ecrire une fonction `marchehasard1.m`, prenant comme variables d'entrée le nombre de pas de temps N considéré et la probabilité p, et retournant comme variable de sortie la position X_N, et traçant optionnellement la trajectoire du marcheur, c'est à dire X_n en fonction de n.
- Dans une fonction `testmarchehasard1.m`, faire une boucle appelant `marchehasard1` un grand nombre M de fois (avec $M \gg N$) pour les mêmes valeurs de p et N, et tracer l'histogramme (en utilisant `hist`) des valeurs de X ainsi obtenues.

Il sera utile de faire plusieurs simulations pour pouvoir interpréter le résultat. On pourra interpréter le résultat en utilisant le théorème de la limite centrale.

2.3 Variables aléatoires discrètes

Le but est ici de simuler numériquement une suite de N nombres pseudo-aléatoires, prenant des valeurs $x_1, \dots x_K$, avec probabilités $p_1, \dots p_K$ fixées.

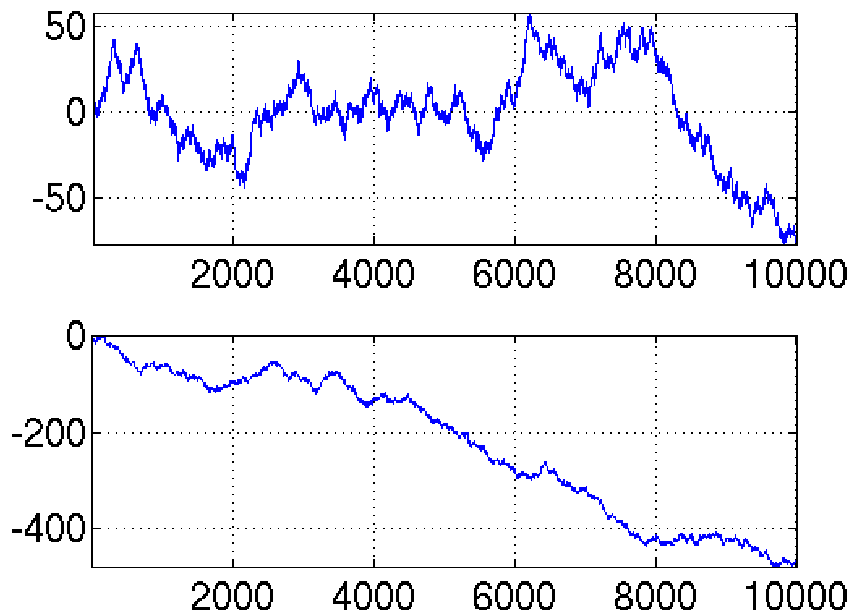

Fig. 2.1 – Deux trajectoires de marche au hasard 1D : position du marcheur en fonction du temps (10 000 pas). Haut : $p = 0.5$; bas : $p = 0.48$.

Exercice 2.13 Générer pour cela une fonction `VAdiscrete.m`, prenant comme variables d'entrée l'ensemble des valeurs possibles $X = \{x_1, \ldots x_K\}$ et le vecteur de probabilités correspondant $P = \{p_1, \ldots p_K\}$, ainsi que le nombre N de valeurs pseudo-aléatoires désirées. La variable de sortie sera la suite de N valeurs ainsi générées.

On s'appuiera sur le générateur de nombres pseudo-aléatoires `rand`, qui fournit des nombres pseudo-aléatoires U uniformément distribués entre 0 et 1, et on remarquera que $p_k = P(U \in [p_{k-1}, p_{k-1} + p_k[$ (avec la convention $p_0 = 0$).

La fonction effectuera les opérations suivantes :

- Vérification que la somme des probabilités vaut 1
- Vérification que X et P sont de même taille
- Génération des nombres pseudo-aléatoires.
- Calcul des fréquences des nombres obtenus, et comparaison (par exemple, par l'erreur relative) avec les probabilités théoriques.

Pour ce qui concerne le dernier point, on pourra utiliser l'instruction

```
>> sum(Z == x)/length(Z)
```

où Z est le nombre généré, et x une valeur donnée... à condition de la comprendre.

2.4 Devoir

Les exercices suivants sont à faire par binôme, et à rendre sous forme d'un fichier "archive", à envoyer à l'adresse `torresan@cmi.univ-mrs.fr`. Pour cela, sous unix/linux, créer un répertoire nommé `nom_prenom_tp2`, et copier tous les fichiers `*.*` dans ce répertoire. Ensuite, dans le répertoire supérieur, effectuer `tar -zcvf nom_prenom_tp2.tgz nom_prenom_tp2/*`. Alternativement, en utilisant le gestionnaire de fenêtres (généralement gnome ou kde), cliquer "droit" sur le répertoire et choisir un format d'archivage et de compactage (de préférence, `.zip`, ou `.tar.gz` (ou `.tgz`, qui est le même format).
Sous Windows, on pourra utiliser un logiciel de compactage (winzip, winrar,7zip,...) pour créer l'archive.

Exercice 2.14 Implémenter une fonction `matmult.m` prenant en entrée deux matrices A et B telles que le produit matriciel AB soit bien défini, testant leurs dimensions (et retournant une erreur si AB n'existe pas) et effectuant le produit matriciel en utilisant des boucles.

Exercice 2.15 Implémenter une fonction permettant de comparer les performances de la fonction précédente et de la multiplication matricielle de Matlab . Cette fonction génèrera des matrices carrées aléatoires de tailles croissantes, les multipliera en utilisant les deux méthodes, et enregistrera le temps de calcul dans les deux cas (en utilisant l'instruction `cputime`). Les temps de calcul seront stockés dans deux tableaux, et tracés dans une fenêtre graphique.

Exercice 2.16 (optionnel) Finaliser la fonction `testmarchehasard1.m` ci-dessus, en superposant à l'histogramme obtenu une Gaussienne de moyenne et variance égales aux valeurs prédites par le théorème de la limite centrale. Alternativement, utiliser la fonction `subplot` pour partitionner la fenêtre graphique en deux sous-fenêtres, tracer l'histogramme à gauche, et la densité théorique à droite.

Exercice 2.17 On s'intéresse maintenant à une marche aléatoire dans le plan. Partant du point de coordonnées $X_0 = (0, 0)$, on génère une suite de points dans le plan de la façon suivante : $X_{n+1} = X_n + (u, v)$ où (u, v) est un vecteur pseudo-aléatoire valant $(1, 0)$, $(-1, 0)$, $(0, 1)$ et $(0, -1)$ avec probabilités égales à $1/4$.

Ecrire une fonction `marchehasard2.m`, qui génère une telle trajectoire. On écrira tout d'abord une fonction générant les vecteurs (u, v) ci-dessus. Cette fonction sera appelée par `marchehasard2` à l'intérieur d'une boucle.

Exercice 2.18 (optionnel) On pourra également compléter cette fonction en ajoutant d'autres affichages graphiques, par exemple en utilisant la fonction `plot3`. On peut également utiliser la fonction `subplot` pour partitionner la fenêtre graphique en deux sous-fenêtres, et tracer la trajectoire en 2D à gauche, et la trajectoire 3D à droite... ou faire plein d'autres choses...

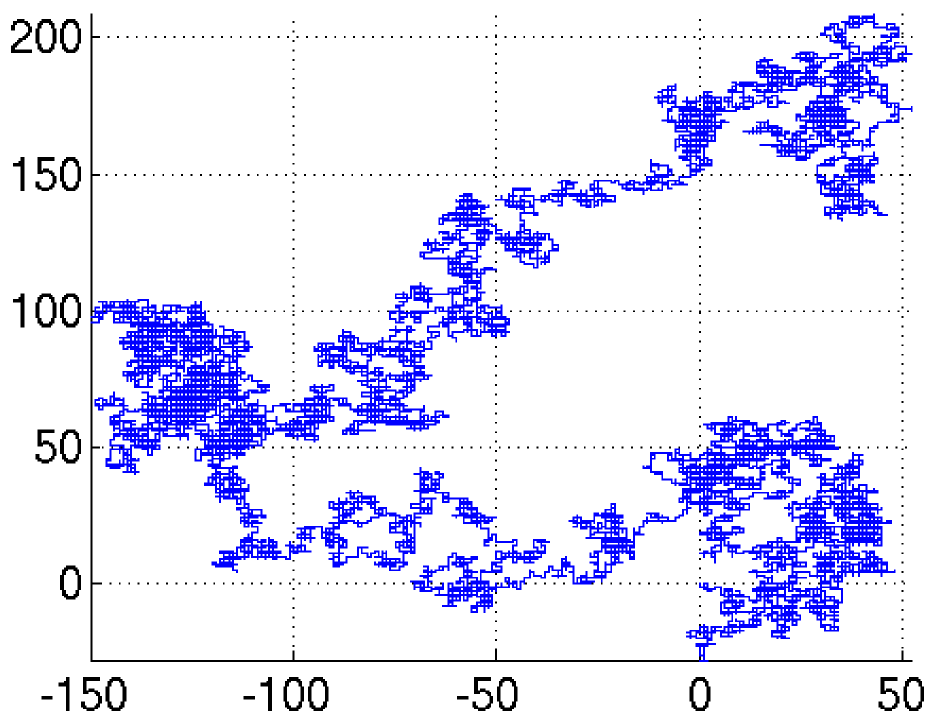

Fig. 2.2 – Trajectoire de marche au hasard 2D : traces de pas du marcheur dans le plan (20 000 pas).

3 Algèbre linéaire

L'objectif de cette troisième séance est d'une part d'approfondir les aspects de Matlab et Octave liés à l'algèbre linéaire, et de voir (ensuite) une application à la régression linéaire.

3.1 Généralités

3.1.1 Vectorisation

Matlab est un langage spécialement adapté à l'algèbre linéaire. Ainsi de nombreuses opérations (telles que par exemple la multiplication matricielle) sont implémentées, et sont beaucoup plus efficaces que si on les reprogramme en utilisant des boucles. Par exemple, la fonction suivante `testmatvectmult.m` compare deux versions de la multiplication matrice-vecteur (les instructions `tic` et `toc` permettent d'avoir une estimation du temps de calcul écoulé entre le `tic` et le `toc`).

```
function [W1,W2] = testmatvectmult(A,V)
% testmatvectmult : teste la multiplication matrice-vecteur
% usage : [W1,W2] = testmatvectmult(A,V)
% Arguments : matrice A et vecteur V, multipliables

[m_A,n_A] = size(A) ;
[m_V,n_V] = size(V) ;
if ~(n_V==1)
     error('V doit etre un vecteur (colonne)') ;
end
if n_A~=m_V
     error('matrice et vecteur non multipliables') ;
end

tic
W1 = zeros(m_A,1) ;
for m = 1 :m_A
     for k = 1 :n_A
          W1(m) = W1(m) + A(m,k)*V(k) ;
     end
end
fprintf('Multiplication avec boucle : ')
toc

tic
W2 = A*V ;
fprintf('Multiplication vectorielle : ')
toc
```

Exercice 3.1 Télécharger cette fonction sur le site du cours, et la tester sur des matrices de tailles 10, 100, 200,...

3.1.2 Décompositions de matrices

Il existe un certain nombre d'algorithmes permettant de décomposer une matrice en produit de matrices plus simples. On peut par exemple citer

- la décomposition QR (fonction `qr`), qui décompose une matrice en un produit d'une matrice triangulaire supérieure R et d'une matrice unitaire Q,
- la décomposition LU (fonction `lu`), qui décompose en un produit d'une matrice triangulaire supérieure U et d'une matrice L, produit d'une matrice triangulaire inférieure et d'une matrice de permutation (ne pas confondre ces dernières avec les parties triangulaires supérieure et inférieure obtenues avec `tril` et `triu`),
- la décomposition de Cholesky (fonction `chol`), qui factorise une matrice définie positive A en un produit $A = LL^*$ où L est triangulaire inférieure.

Exercice 3.2
- Tester ces fonctions sur de petites matrices.
- Ecrire une fonction `QRresol.m` prenant comme variable d'entrée une matrice carrée $A \in \mathcal{M}(N)$ et un vecteur $y \in \mathcal{M}(N, 1)$, et utilisant la décomposition QR pour
  - calculer le déterminant de A et donc tester si A est inversible
  - si A est inversible, résoudre le système $Ax = y$ (qui devient donc $Rx = Q^{-1}y$).

  On rappelle que le déterminant d'une matrice unitaire est égal à 1, et que son inverse est égal à sa conjuguée hermitienne (complexe conjuguée de la transposée). On aura besoin d'expliciter l'inverse d'une matrice triangulaire supérieure, et de le coder (en utilisant une boucle).

3.1.3 Déterminants

L'instruction `det` permet de calculer le déterminant d'une matrice carrée. Il est aussi instructif de calculer ce déterminant directement. On peut pour celà utiliser l'expression en fonction des cofacteurs. On rappelle que les cofacteurs d'une matrice $A = \{a_{k\ell}\} \in \mathcal{M}_N$ sont définis par

$$c_{k\ell}(A) = (-1)^{k+\ell} \det(A^{(k,\ell)}) \ ,$$

où $A^{(k,\ell)}$ est la matrice réduite, obtenue à partir de A en lui enlevant la k-ième ligne et la ℓ-ième colonne. Le déterminant s'exprime alors sous la forme

$$\det(A) = \sum_{\ell} a_{k_0\ell} c_{k_0\ell}(A) = \sum_{k} a_{k\ell_0} c_{k\ell_0}(A) \ ,$$

où ℓ_0 (resp. k_0) est une colonne (resp. ligne) fixée.

Exercice 3.3 Ecrire une fonction `cofact.m`, prenant comme variables d'entrée une matrice carrée A (à N lignes et N colonnes), et deux entiers $1 \leq k, \ell \leq N$, et retournant le cofacteur correspondant $c_{k\ell}(A)$.

Exercice 3.4 Ecrire une fonction `mondet.m`, calculant le déterminant d'une matrice carrée en utilisant la fonction `cofact.m` ci-dessus.

Exercice 3.5 Ecrire une fonction `cofact2.m` et une fonction `mondet2.m` s'appelant l'une l'autre. Comparer le résultat obtenu avec `det` et `mondet` sur des matrices de tailles 3,4,5,6,... Attention, `mondet2` devient vite inefficace lorsque N grandit...

3.1.4 Rang et noyau

L'instruction `rref` analyse l'espace image d'une matrice, qu'elle retourne sous forme réduite. Par exemple, la séquence d'instructions

```
>> A = magic(4) ;% Génère une matrice particulière
>> [R,jb] = rref(A) ;% réduit la matrice
>> Rang = length(jb) ;
>> Base = R(jb) ;
```

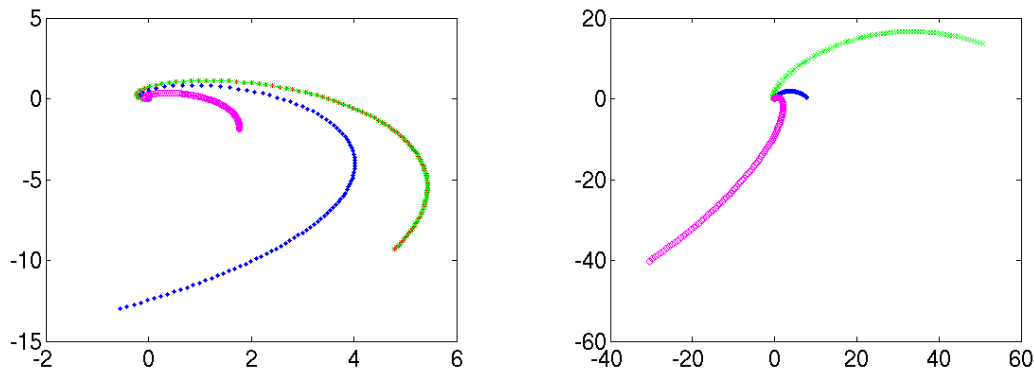

FIG. 3.1 – Trois trajectoires solutions du système $X'(t) = AX(t)$, pour trois conditions initiales différentes, et deux matrices $A = A_1$ (gauche) et $A = A_2$ (droite). Les valeurs du temps sont prises sur une échelle logarithmique.

donne une évaluation du rang de A, et une matrice (ici appelée `Base`) dont les colonnes forment une base de l'image de A. Une base orthonormale de l'image peut aussi être directement obtenue grace à `orth`, et l'instruction `rank` donne le rang.

L'instruction `null` retourne une base orthonormale du noyau d'une matrice carrée.

3.1.5 Inversion matricielle et systèmes linéaires

On a déjà rencontré les fonctions concernant la diagonalisation et l'inversion des matrices. L'inverse d'une matrice carrée est obtenue via la fonction `inv`. Se documenter sur cette instruction (`doc inv`). Lorsque la matrice est proche d'être singulière, un message d'erreur s'affiche dans la fenêtre de commande. Voir aussi l'instruction `cond`. Les fonction `mldivide` et `rldivide` retournent des inverses à gauche et à droite d'une matrice, pas nécessairement carrée[1]. Attention : MATLAB effectuant des opérations numériques, avec une précision donnée, il est capable d'« inverser » des matrices non inversibles... mais donnera un message d'erreur dans ce cas là. De là, il est possible de résoudre des systèmes linéaires à l'aide de ces instructions pré-définies

EXERCICE 3.6 Générer trois matrices et deux vecteurs en utilisant la séquence d'instructions suivante :

```
>> A = [1,2,3 ;4,5,6 ;7,8,10] ;
>> B = A(1 :2, :) ;
>> C = A( :,1 :2) ;
>> D = A(1 :2,1 :2) ;
>> x = [1 ;2 ;3] ;
>> y = x(1 :2) ;
```

et interpréter ces instructions au vu des résultats obtenus.

Parmi les systèmes matriciels suivants, quels sont ceux qui admettent des solutions ? une solution unique ?

$$x = Az \; ; \quad x = Cz \; ; \quad y = Bz \; ; \quad y = Dz$$

Résoudre ces systèmes en utilisant l'instruction `inv` ou `mldivide`. Utiliser l'aide en ligne pour interpréter le résultat obtenu.

3.1.6 Diagonalisation

La fonction `eig`, qu'on a déjà vue, permet de calculer à la fois les vecteurs propres et valeurs propres d'une matrice carrée donnée. On rappelle que la syntaxe est la suivante :

```
>> [VectPropres, ValPropres] = eig(A) ;
```

[1] à ne pas confondre avec les instructions `ldivide` et `rdivide`, qui effectuent des divisions "point par point" de matrices de même taille.

et que `ValPropres` est une matrice diagonale, `VectPropres` étant une matrice dont les colonnes sont les vecteurs propres correspondant aux valeurs propres trouvées, dans le même ordre.

EXERCICE 3.7 1. partant de la matrice A ci-dessous

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 5 \end{pmatrix}$$

résoudre numériquement le système différentiel

$$\dot{x}(t) = Ax(t) \ , \qquad x(0) = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

en utilisant la diagonalisation de A, et le fait que la solution s'exprime de façon explicite :

$$x(t) = Pe^{tD}P^{-1}x(0) \ .$$

On pourra par exemple tracer le résultat obtenu.

2. Résoudre le problème précédent de façon systématique : écrire une fonction `masolexp.m` prenant comme variables d'entrée la matrice A, la condition initiale, et un ensemble de valeurs de t, et retournant comme variables de sortie les coordonnées de la solution $x(t)$.
3. Dans le cas bidimensionnel, compléter cette fonction en traçant les valeurs ainsi obtenues comme des points dans le plan. On pourra essayer les matrices suivantes

$$A_1 = \begin{pmatrix} 1 & 1 \\ -1 & 1 \end{pmatrix} \ , \qquad A_2 = \begin{pmatrix} 1 & 1 \\ -1 & 2 \end{pmatrix} \ ,$$

et il est conseillé de prendre des valeurs du temps sur une échelle logarithmique (par exemple avec une instruction du type `T=log(linspace(0.01,1,100))`, qui prend les logarithmes de 100 valeurs régulièrement espacées entre 0.01 et 1).

Un exemple de trajectoire avec la matrice A_1 (et trois conditions initiales différentes) se trouve en figure 3.1.

3.2 Pseudo-inverse

3.2.1 Orthogonalisation et complétion de Gram-Schmidt

Un résultat classique d'algèbre linéaire montre qu'étant donnés n vecteurs orthogonaux deux à deux et normalisés dans un espace Euclidien (ou de Hilbert) de dimension $N > n$, il est toujours possible de trouver $N - n$ vecteurs normalisés de ce même espace tels que la famille complète des N vecteurs forme une base orthonormée de l'espace considéré.
La procédure de Gram-Schmidt fournit un procédé constructif permettant d'obtenir ces vecteurs. Notons $v_1, \dots v_n \in E$ les n premiers vecteurs. Soit $w \in E$ un vecteur n'appartenant pas au sous-espace de E engendré par $\{v_1, \dots v_n\}$. En posant

$$\tilde{v}_{n+1} = w - \sum_{k=1}^{n} \langle w, v_k \rangle v_k$$

et

$$v_{n+1} = \frac{\tilde{v}_{n+1}}{\|\tilde{v}_{n+1}\|}$$

il est facile de vérifier que la famille $\{v_1, \dots v_{n+1}\}$ est une famille orthonormée dans E.
L'orthogonalisation de Gram-Schmidt utilise ce principe autant de fois qu'il le faut pour compléter la famille initiale et engendrer ainsi une base orthonormée de E.

La version matricielle de cette construction est la suivante. En mettant en colonne les composantes des n vecteurs initiaux dans la base canonique, on obtient une matrice M à N lignes et n colonnes. Cette matrice

est isométrique, c'est à dire telle que $M^*M = I_n$; elle n'est par contre pas unitaire, car $MM^* \neq I_N$. La procédure de Gram-Schmidt permet alors de concaténer itérativement $N - n$ colonnes à M, de sorte à obtenir une matrice U unitaire, c'est à dire telle que $U^*U = UU^* = I_N$.

Exercice 3.8 Ecrire une fonction `GramSchmidtComplete.m` prenant comme variable d'entrée une matrice isométrique $N \times n$ (avec $n < N$), et retournant une matrice unitaire complétée à la Gram-Schmidt. La fonction vérifiera que la matrice d'entrée est bien isométrique.

Pour utiliser la procédure de Gram-Schmidt, il est nécessaire de savoir générer à chaque itération un vecteur n'appartenant pas au sous espace engendré par les vecteurs initiaux. L'instruction `randn` permettra de générer un vecteur qui vérifiera presque sûrement cette propriété.

3.2.2 Décomposition en valeurs singulières

Théorie De façon générale, la théorie de la diagonalisation s'applique aux endomorphismes (et donc aux matrices carrées). Dans le cas de morphismes entre espaces vectoriels différents, il reste possible d'effectuer des opérations similaires, même lorsque les espaces sont de dimensions différentes.

Théorème 3.1 *Soit $M \in \mathcal{M}_{\mathbb{K}}(m, n)$ une matrice $m \times n$ à coefficients dans le corps $\mathbb{K}$, avec $\mathbb{K} = \mathbb{R}$ ou $\mathbb{K} = \mathbb{C}$. Alors il existe une factorisation de la forme :*

$$M = U\Sigma V^* \ ,$$

où U est une matrice unitaire $m \times m$ sur $\mathbb{K}$, Σ est une matrice $m \times n$ dont les coefficients diagonaux sont des réels positifs ou nuls et tous les autres sont nuls (c'est donc une matrice diagonale dont on impose que les coefficients soient positifs ou nuls), et V^* est la matrice adjointe de V, matrice unitaire $n \times n$ sur $\mathbb{K}$. On appelle cette factorisation la décomposition en valeurs singulières de M.

- La matrice V contient un ensemble de vecteurs de base orthonormés pour M, dits *vecteurs d'entrée* ;
- La matrice U contient un ensemble de vecteurs de base orthonormés pour M, dits *vecteurs de sortie* ;
- La matrice Σ contient les *valeurs singulières* de la matrice M.

3.2.3 Construction de la SVD :

La construction de cette matrice peut se faire de la façon suivante : on considère la matrice

$$A = M^*M \ ,$$

qui est par construction semi-définie positive, donc Hermitienne. Elle est donc diagonalisable, avec des valeurs propres réelles positives ou nulles, et il existe une matrice $D \in \mathcal{M}(n, n)$ et une matrice de passage $V \in \mathcal{M}(n, n)$ telles que

$$A = VDV^{-1} \ .$$

V peut être choisie <u>unitaire</u>, de sorte que $V^{-1} = V^*$. La matrice diagonale D prend quant à elle la forme

$$D = \begin{pmatrix} D_1 & 0 \\ 0 & 0 \end{pmatrix} \ ,$$

$D_1 \in \mathcal{M}(r, r)$ étant une matrice diagonale dont les termes diagonaux sont strictement positifs (r est le rang de A et D). Il est usuel d'ordonner les valeurs diagonales de D par ordre décroissant.
On note $V_1 \in \mathcal{M}(n, r)$ la matrice dont les colonnes sont les vecteurs propres de A de valeur propre non nulle (les valeurs propres étant rangées en ordre décroissant), et $V_2 \in \mathcal{M}(n, n - r)$ la matrice formée des autres vecteurs propres. Notons que V_1 n'est pas unitaire : on a $V_1^* V_1 = I_r$ mais $V_1 V_1^* \neq I_n$.
Posons maintenant (D_1 étant inversible)

$$U_1 = MV_1 D_1^{-1/2} \in \mathcal{M}(m, r)$$

On a alors $U_1^* U_1 = I_r$, mais $U_1 U_1^* \neq I_m$. Il est par contre possible de montrer qu'en complétant U_1 par $m - r$ colonnes orthogonales deux à deux, et orthogonales aux colonnes de U_1, on obtient une matrice unitaire U. On montre alors

$$U\Sigma = MV \ ,$$

où Σ s'obtient en complétant D_1 par des zéros, d'où le résultat.

Remarque : cette démonstration de l'existence de la décomposition est constructive, elle fournit un algorithme de calcul de la svd qu'on va programmer plus loin. Ceci dit, cet algorithme est connu pour être numériquement instable (en présence de petites valeurs singulières), c'est pourquoi les bibliothèques lui préfèrent des algorithmes plus élaborés.

EXERCICE 3.9 Ecrire une fonction `masvd.m`, prenant en entrée une matrice M, et calculant sa décomposition en valeurs singulières.
- Diagonaliser la matrice $A = M^*M$. Réorganiser le résultat (les matrices D et V) de sorte que les valeurs propres soient classées par ordre décroissant (utiliser la fonction `sort`). Tracer les valeurs propres classées (en utilisant l'instruction `bar` plutôt que `plot`).
- Extraire la matrice V_1 et la matrice D_1, et en déduire la matrice U_1.
- Compléter U_1 en une matrice unitaire U : on pourra pour cela utiliser la procédure d'orthogonalisation de Gram-Schmidt. Compléter Σ.
- La fonction retournera les matrices U, V, et Σ.

Remarque : L'analyse en composantes principales présente de grosses similarités avec la SVD. En effet, étant donné un tableau de données X, on effectue une diagonalisation de la matrice X^*X (ou XX^*), après avoir centré les lignes ou les colonnes de X. L'opération finale de l'ACP est une projection des données sur les sous-espaces engendrés par un certain nombre de vecteurs propres.

3.2.4 Pseudo-Inverse

Soient E et F deux espaces vectoriels, et soit $f \in \mathcal{L}(E, F)$ une application linéaire de E dans F. Lorsque f n'est pas inversible, la notion de pseudo-inverse fournit un substitut intéressant.

Théorie

DÉFINITION 3.1 *Une application linéaire $g \in \mathcal{L}(F, E)$ est appelée pseudo-inverse de f lorsqu'elle satisfait les deux conditions*

$$f \circ g \circ f = f \qquad g \circ f \circ g = g$$

On montre dans ce cas les propriétés suivantes
- E est la somme directe du noyau de f et de l'image de g
- F est la somme directe du noyau de g et de l'image de f
- f induit un isomorphisme de l'image de g sur l'image de f
- g induit un isomorphisme de l'image de f sur l'image de g, qui est l'inverse de l'isomorphisme précédent.

Il s'agit bien d'une extension de la notion d'inverse dans la mesure où si f admet effectivement un inverse, celui-ci est aussi un, et même l'unique, pseudo-inverse de f.

Interprétation La pseudo-inverse d'une matrice $A \in \mathcal{M}(m, n)$ peut être interprétée de façon assez intuitive en termes de systèmes linéaires. En considérant le système linéaire

$$AX = Y \ ,$$

$A^\dagger Y$ est, parmi les vecteurs X qui minimise l'erreur en moyenne quadratique $\|Y - AX\|$, celui dont la norme $\|X\|$ est minimale.
- Si le rang de A est égal à m, alors AA^* est inversible, et $A^\dagger = A^*(AA^*)^{-1}$
- Si le rang de A est égal à n, alors A^*A est inversible, et $A^\dagger = (A^*A)^{-1}A^*$ Si le rang de A est égal à $m = n$, alors A est inversible, et $A^\dagger = A^{-1}$.

Construction des pseudo-inverses La description géométrique qui vient d'être donnée pour les pseudo-inverses est une propriété caractéristique de ceux-ci. En effet, si on introduit des supplémentaires K du noyau de f (dans E) et I de l'image de f (dans F), il est possible de leur associer un unique pseudo-inverse g de f. Les restrictions de g aux espaces I et image de f sont parfaitement définies : application nulle sur I, et inverse de l'isomorphisme induit par f sur K, sur $Im(f)$. Les deux propriétés de pseudo-inverses sont alors facilement vérifiées en séparant selon les couples d'espaces supplémentaires. Cette construction montre qu'en général il n'y a pas unicité du pseudo-inverse associé à une application linéaire.

PROPOSITION 3.1 *Supposons maintenant que E et F soient deux espaces Hermitiens (Euclidiens dans le cas réel). Soit A la matrice de f par rapport à deux bases données de E et F, soit $A^\dagger$ la matrice du pseudo-inverse g. Alors $A^\dagger$ est l'unique matrice qui satisfait les conditions :*

1. $AA^\dagger A = A$
2. $A^\dagger AA^\dagger = A^\dagger$
3. $AA^\dagger$ *est une matrice Hermitienne :* $(AA^\dagger)^* = AA^\dagger$.
4. $A^\dagger A$ *est une matrice Hermitienne :* $(A^\dagger A)^* = A^\dagger A$.

où on a noté M^* la conjuguée Hermitienne de la matrice M.

Calcul de la pseudo-inverse Notons k le rang de $A \in \mathcal{M}(m,n)$, une matrice $m \times n$. A peut donc être décomposée sous la forme

$$A = BC \ ,$$

où $B \in \mathcal{M}(m,k)$ et $C \in \mathcal{M}(k,n)$. On a alors

PROPOSITION 3.2 *Avec les notations ci-dessus, la pseudo-inverse de A s'écrit*

$$A^\dagger = C^*(CC^*)^{-1}(B^*B)^{-1}B^*$$

Si $k = m$, B peut être la matrice identité. De même, si $k = n$, C peut être l'identité. Ainsi, la formule se simplifie.
En pratique, la pseudo-inverse se calcule à partir de la décomposition en valeurs singulières :

$$A = U\Sigma V^* \ ,$$

où Σ est diagonale. On a alors

$$A^\dagger = V\Sigma^\dagger U^* \ ,$$

où $\Sigma^\dagger$ est obtenue à partir de Σ par transposition, et remplacement des éléments non nuls par leur inverse.

EXERCICE 3.10 En utilisant soit votre fonction `masvd`, soit la fonction `svd`, construire une fonction `mapseudoinverse.m`.

3.3 Devoir

Les devoirs sont à rendre par binôme. Finaliser les fonctions `masolexp.m`, `GramSchmidtComplete.m`, `masvd.m` et `mapseudoinverse.m` demandées dans les exercices des sections précédentes.
Les devoirs de cette section sont à transmettre sous forme d'archive, par exemple sous forme de fichier `noms_tp3.tgz`, `nom_tp3.rar` ou `noms_tp3.zip`. L'archive devra contenir les fichiers des programmes, mais aussi un petit compte-rendu décrivant le travail qui a été fait, le cas échéant ce qui n'a pas pu être fait ou terminé, ainsi que quelques explications sur l'utilisation des fonctions. On pourra aussi ajouter des fichiers de graphiques (format `jpg` par exemple).

4 Interpolation, régression

L'interpolation est une opération mathématique permettant de construire une courbe à partir de la donnée d'un nombre fini de points, ou une fonction à partir de la donnée d'un nombre fini de valeurs. La solution du problème d'interpolation passe par les points prescrits, et, suivant le type d'interpolation, il lui est demandé de vérifier des propriétés supplémentaires.
Le type le plus simple d'interpolation est l'interpolation linéaire, qui consiste à "joindre les points" donnés. Il s'agit d'une procédure locale, à l'inverse de l'interpolation globale telle que l'interpolation polynômiale.

L'interpolation doit être distinguée de l'approximation de fonction, aussi appelée régression, qui consiste à chercher la fonction la plus proche possible, selon certains critères, d'une fonction donnée. Dans le cas de la régression, il n'est en général plus imposé de passer exactement par les points donnés initialement. Ceci permet de mieux prendre en compte le cas des erreurs de mesure, et c'est ainsi que l'exploitation de données expérimentales pour la recherche de lois empiriques relève plus souvent de la régression linéaire, ou plus généralement de la méthode des moindres carrés.

4.1 Interpolation

4.1.1 Inperpolation polynômiale : Lagrange

L'interpolation polynomiale consiste à utiliser un polynôme unique (et non des segments comme on le verra plus loin), de degré aussi grand que nécessaire, pour estimer localement l'équation représentant la courbe afin de déterminer la valeur entre les échantillons. Comme on le verra, l'interpolation polynômiale a tendance à devenir instable lorsque le degré du polynôme croît.
Le prototype de l'interpolation polynômiale est l'interpolation de Lagrange, décrite et analysée ci-dessous.

Théorème 4.1 *Soient* $\{(x_k, y_k), k = 0 \dots n\}$ $n+1$ *points donnés, tels que les* x_k *soient tous deux à deux différents. Il existe un et un seul polynôme de degré* n*, noté* p_n*, tel que* $p_n(x_k) = y_k$ *pour tout* $k = 0, \dots n$*. Ce polynôme est de la forme*

$$p_n(x) = \sum_{k=0}^{n} y_k \ell_k(x) \ , \tag{4.1}$$

où les ℓ_k *sont des polynômes de degré* n *donnés par*

$$\ell_k(x) = \prod_{j \neq k} \frac{x - x_j}{x_k - x_j} \ . \tag{4.2}$$

Ce résultat peut se démontrer de différentes façons. La plus simple est sans doute d'effectuer le calcul explicite : le polynôme p recherché doit satisfaire $p(x_k) = y_k$ pour tout $k = 0, \dots n$, ce qui conduit au système

$$\sum_{j=0}^{n} a_j x_k^j = y_k \ , \quad k = 0, \dots n \ ,$$

qui s'écrit sous forme matricielle

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \dots & x_0^n \\ 1 & x_1 & x_1^2 & \dots & x_1^n \\ 1 & x_2 & x_2^2 & \dots & x_2^n \\ \vdots & \vdots & \ddots & \vdots & \\ 1 & x_n & x_n^2 & \dots & x_n^n \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

Ce système admet une solution unique si et seulement si le déterminant de cette matrice, appelé *déterminant de Vandermonde* est non-nul. On peut alors utiliser le résultat suivant

LEMME 4.1 *Le déterminant de Vandermonde est donné par*

$$\begin{vmatrix} 1 & x_0 & x_0^2 & \dots & x_0^n \\ 1 & x_1 & x_1^2 & \dots & x_1^n \\ 1 & x_2 & x_2^2 & \dots & x_2^n \\ \vdots & \vdots & \ddots & \vdots & \\ 1 & x_n & x_n^2 & \dots & x_n^n \end{vmatrix} = \prod_{j<k} (x_k - x_j) \tag{4.3}$$

Comme nous avons supposé que les x_k sont deux à deux disjoints, nous sommes assurés de l'existence et unicité de la solution. Finalement, il suffit de vérifier que la solution proposée satisfait bien les conditions exigées, et l'unicité fait le reste. ♠

MATLAB n'implémente pas de fonction effectuant explicitement l'interpolation de Lagrange. Ceci dit, il est possible d'utiliser une alternative, basée sur l'approximation par moindres carrés, qu'on verra plus loin. On se contente à ce point du résultat suivant, dont la preuve est immédiate.

PROPOSITION 4.1 *Soient* $\{(x_k, y_k), k = 0 \dots n\}$ $n+1$ *points donnés, tels que les* x_k *soient tous deux à deux différents. Soit* q_n *le polynôme de degré* n *qui minimise l'erreur d'approximation en moyenne quadratique*

$$q_n = \arg\min_q \sum_{k=0}^{n} |y_k - q(x_k)|^2 \ .$$

Alors $q_n = p_n$, *le polynôme d'interpolation de Lagrange.*

La fonction `polyfit` de MATLAB effectue l'interpolation polynômiale par moindres carrés. On peut donc l'utiliser pour résoudre le problème d'interpolation de Lagrange, et illustrer ses caractéristiques. On utilise ensuite la fonction `polyval` pour évaluer le polynôme ainsi obtenu en des points bien choisis.

EXERCICE 4.1 *Se renseigner sur les fonctions* `polyfit` *et* `polyval`. *Etudier l'exemple suivant, qui illustre l'utilisation de ces fonctions.*

```
>> n = 4 ;
>> x = rand(n+1,1) ;          Génération de 5 abscisses tirées au hasard
>> x = sort(x) ;              Réarrangement dans l'ordre croissant
>> y = randn(n+1,1) ;         Génération d'ordonnées aléatoires
>> P = polyfit(x,y,n) ;       Calcul du polynôme interpolateur
>> xmin = min(x) ; xmax = max(x) ;
>> xx = linspace(xmin,xmax,50) ;
>> yy = polyval(P,xx) ;
>> plot(xx,yy) ;
>> hold on ;
>> plot(x,y,'.r') ;
```

Effectuer cet exercice plusieurs fois (en changeant aussi la valeur de n*). Sauvegarder quelques figures significatives, et commenter ce que l'on observe dans un fichier* compte rendu.

Un exemple se trouve en Figure 4.1. On peut observer que le polynôme interpolant peut s'éloigner très significativement des données. Ce phénomène est à rapprocher du *phénomène de Runge*, que l'on rencontre lorsque l'on s'intéresse aux propriétés de convergence de l'approximation d'une fonction par ses polynômes de Lagrange.

Une question naturelle pour le mathématicien est d'analyser la précision de l'interpolation dans le cas d'une fonction connue : en d'autres termes, étant donnée une fonction f, quelques valeurs ponctuelles $f(x_k)$, et le polynôme p_n de degré n obtenu par interpolation de lagrange à partir de celles-ci, quelle est la différence entre f et p_n ? On déduit facilement du théorème de Rolle le résultat suivant.

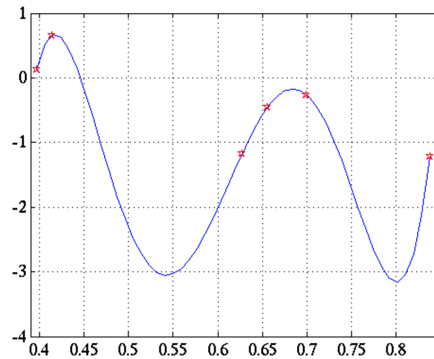

FIG. 4.1 – Interpolation polynômiale : polynôme de degré 5 ; les données initiales sont représentées par des étoiles.

PROPOSITION 4.2 *Soit $f \in C^{n+1}$ une fonction sur un intervalle $I = [a, b]$, et soit p_n le polynôme de Lagrange associé aux données $(x_0, y_0), \dots (x_n, y_n)\}$. Alors, $\forall x$, il existe ξ_x tel que*

$$f(x) - p_n(x) = \frac{f^{(n+1)}(\xi_x)}{(n+1)!}\pi_{n+1}(x)$$

où π_{n+1} est le polynôme de degré $n+1$ défini par $\pi_{n+1}(x) = \prod_{k=0}^{n}(x - x_k)$.

On en déduit facilement la borne

$$|f(x) - p_n(x)| \leq \frac{\sup |f^{(n+1)}|}{(n+1)!} \sup |\pi_{n+1}| \ ,$$

de sorte que la précision de l'approximation dépend, comme on peut s'y attendre, de la régularité de f, mais aussi de la position des points x_k. Sans hypothèse supplémentaire, on a en se plaçant dans un intervalle $[a, b]$

$$|\pi_{n+1}(x)| \leq (b-a)^{n+1} \ ,$$

alors qu'en se plaçant dans le cas de points x_k régulièrement espacés dans l'intervalle $[a, b]$ on peut montrer que

$$|\pi_{n+1}(x)| \leq \left(\frac{b-a}{e}\right)^{n+1} \ .$$

Dans un cas comme dans l'autre, cette borne n'est pas très précise.

Si f est une fonction analytique (c'est à dire si sa série entière converge dans un domaine de rayon R), alors la croissance des dérivées successives $f^{(n+1)}(\xi_x)$ en fonction de n peut être estimée, et il est possible de montrer que p_n converge vers f dans un intervalle dont la largeur dépend de R. L'exercice qui suit est un exemple classique montrant la (non) convergence de l'approximation par polynôme de Lagrange.

EXERCICE 4.2 On considère la fonction $u_\gamma : [-1, 1] \to \mathbb{R}$ définie par

$$u_\gamma(x) = \frac{1}{x^4 + \gamma} ,$$

où $\gamma \in \mathbb{R}^+$ est un réel positif fixé, et on s'intéresse à ses approximations polynômiales par interpolation de Lagrange.

Ecrire une fonction `erreurinterpol.m`, prenant comme variable d'entrée la valeur de γ et un ensemble de points d'échantillonnage `x`, et évaluant la précision de l'approximation par polynôme interpolateur de Lagrange :

1. Calcul des valeurs de u_γ sur les points `x`.
2. Evaluation du polynôme interpolateur
3. Génération d'un vecteur de points régulièrement espacés `xx` (utiliser la fonction `linspace`), et évaluation de u_γ en ces points.
4. Evaluation de l'approximation de la fonction u_γ en ces points
5. Calcul de l'erreur relative : norme de l'erreur divisé par la norme de la référence $\|u_\gamma\|$.

La fonction retournera l'erreur relative en variable de sortie.

On effectuera plusieurs essais, avec des points `x` tirés aléatoirement entre -1 et 1, puis régulièrement espacés ; on fera quelques courbes significatives, avec diverses valeurs de γ et n, qu'on commentera dans le compte rendu.

4.1.2 Polynômes par morceaux

Interpolation linéaire par morceaux Dans le cas d'une interpolation linéaire, on constitue une courbe d'interpolation qui est une succession de segments. Entre deux points $p_1 = (x_1, y_1)$ et $p_2 = (x_2, y_2)$, l'interpolation est donnée par la formule suivante

$$y = p \cdot (x - x_1) + y_1$$

où la pente p s'exprime comme

$$p = \frac{y_2 - y_1}{x_2 - x_1}$$

EXERCICE 4.3 Programmer une fonction MATLAB `interpollin.m`, prenant comme variables d'entrée les coordonnées $\{(x_0, y_0), \ldots (x_n, y_n)\}$, ainsi qu'une liste de valeurs u_k, calculant une interpolation linéaire par morceaux, la traçant dans une figure, et retournant comme variable de sortie la liste de valeurs interpolées.

On sauvegardera un (ou plusieurs) exemples significatifs de figure, qu'on commentera dans le compte rendu.

Interpolation spline L'interpolation "spline" généralise l'interpolation linéaire par morceaux en remplaçant les morceaux affines par des morceaux polynômiaux. La plus "classique" est l'interpolation par fonctions spline "cubiques", qui sont donc des polynômes de degré 3.

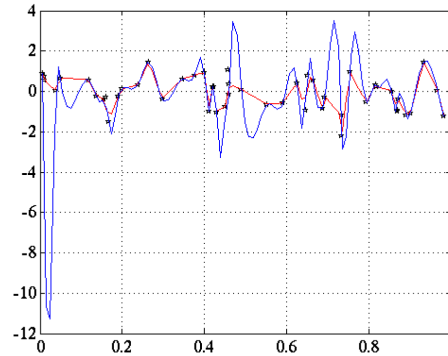

FIG. 4.2 – Interpolation spline : linéaire (rouge) et cubique (bleu) ; les données initiales sont représentées par des étoiles.

DÉFINITION 4.1 *Etant donnés $n+1$ points $\{(x_0, y_0), \dots (x_n, y_n)\}$ (appelés "noeuds"), une fonction interpolante spline cubique passant par ces points est une fonction polynômiale de degré 3 par morceaux*

$$S(x) = \begin{cases} S_0(x), & x \in [x_0, x_1] \\ S_1(x), & x \in [x_1, x_2] \\ \dots & \\ S_{n-1}(x), & x \in [x_{n-1}, x_n] \end{cases}$$

telle que

1. $S(x_k) = f(x_k)$ *pour* $k = 0, \dots n$
2. S *est continue sur les noeuds,* $S_{k-1}(x_k) = S_k(x_k)$*, pour* $k = 1, \dots, n-1$
3. S *est deux fois continûment différentiable ux noeuds :* $S'_{k-1}(x_k) = S_k(x_k)$ *et* $S''_{k-1}(x_k) = S''_k(x_k)$*, pour* $k = 1, \dots, n-1$.

Pour déterminer une telle fonction d'interpolation, on doit déterminer $4n$ paramètres (chaque polynôme de degré 3 est caractérisé par 4 paramètres). Les conditions d'interpolation donnent $n+1$ équations linéaires, et les conditions de continuité en donnent $3n-3$, donc au total $4n-2$ conditions. Les deux conditions restantes sont au choix de l'utilisateur ; par exemple, les splines cubiques "naturelles" supposent

$$S''(x_0) = S''(x_n) = 0 \ .$$

EXERCICE 4.4 *Se documenter sur la fonction* `interp1`*, et tester les commandes suivantes.*

```
>> x = rand(50,1) ;
>> x = sort(x) ;
>> y = randn(size(x)) ;
>> plot(x,y,'.') ;
>> xmin = min(x) ; xmax = max(x) ;
>> x2 = linspace(xmin,xmax,100) ;
>> y2 = interp1(x,y,x2,'linear') ;
>> hold on ; plot(x2,y2,'r') ;
```

Comparer le résultat de l'interpolation linéaire effectuée dans l'exercice 4.3 avec celui de l'interpolation effectuée avec l'option `'linear'` *de* `interp1`*, puis l'option* `'spline'`*.*

Tracer un exemple significatif en comparant l'interpolation linéaire par morceaux et l'interpolation cubique par morceaux. Sauvegarder la figure, et la commenter dans le compte rendu. Un exemple se trouve en Figure 4.2

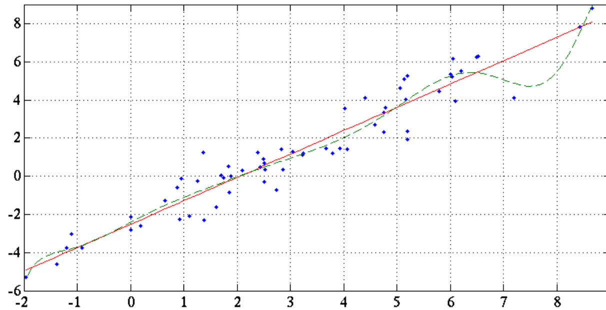

FIG. 4.3 – Régression : linéaire (rouge/trait plein) et polynôme d'ordre 5 (vert/tirets) ; les données initiales sont représentées par des points.

4.2 Régression

Dans le problème de régression, il s'agit encore une fois de décrire des données par une fonction s'en approchant, mais on relâche cette fois la contrainte de "passer par les points exactement" par une contrainte de type "moindres carrés".
On peut là encore faire de la régression linéaire ou polynômiale, ou de la régression polynômiale "par morceaux".

4.2.1 Régression linéaire

Commençons par le cas unidimensionnel. Supposons données des observations $\{(x_0, y_0), \dots (x_n, y_n)\}$, que l'on suppose liées par une relation linéaire

$$y_k = ax_k + b + \epsilon_k \ ,$$

où a et b sont deux réels, et où les ϵ_k représentent un "bruit", c'est à dire une erreur.
Comme on l'a déjà vu, les coefficients a et b peuvent être estimés par moindres carrés, en résolvant le problème

$$\min_{a,b} \sum_k |y_k - ax_k - b|^2 \tag{4.4}$$

En égalant à zéro les dérivées de cette expression par rapport à a et b, on obtient

PROPOSITION 4.3 *La solution du problème (4.4) est donnée par les estimateurs $\hat{a}$ et $\hat{b}$ définis par*

$$\hat{a} = \frac{\sum_k (x_k - \overline{x})(y_k - \overline{y})}{\sum_k (x_k - \overline{x})^2} \ , \quad \hat{b} = \overline{y} - \hat{a}\overline{x} \ ,$$

où

$$\overline{x} = \frac{1}{n+1} \sum_{k=0}^{n} x_k \ , \quad \overline{y} = \frac{1}{n+1} \sum_{k=0}^{n} y_k \ ,$$

Un exemple de régression linéaire se trouve dans la figure 4.3 (avec un exemple correspondant de régression polynômiale, voir plus bas).
Dans une problématique de statistique descriptive, il arrive fréquemment que l'on désire savoir si deux variables sont liées linéairement ou pas. La décision peut se faire via un test sur le coefficient de corrélation de Pearson.

$$\rho = \rho_{xy} = \frac{\sum_{k=1}^{n} (x_k - \bar{x})(y_k - \bar{y})}{(n-1) s_x s_y},$$

où s_x et s_y sont les écarts-types de x et y

$$s_x^2 = \frac{1}{n-1} \sum_{k=1}^{n} (x_k - \bar{x})^2.$$

LEMME 4.2 *Supposons données des observations $\{(x_1, y_1), \dots (x_n, y_n)\}$. Sous l'hypothèse d'indépendance de x et y, le coefficient de Pearson suit une loi de Student-t à $n-1$ degrés de liberté.*

Ainsi, après avoir effectué une régression linéaire, il est possible de tester la significativité de la relation linéaire en comparant la valeur obtenue pour ρ à sa distribution sous hypothèse de décorrélation.

EXERCICE 4.5 Ecrire une fonction `regrlin.m` prenant en entrée deux vecteurs de même dimension, et retournant les nombres a et b estimés, ainsi que l'écart-type de l'erreur estimée $\hat{\epsilon}_k = y_k - \hat{a}x_k - \hat{b}$ et le coefficient de corrélation de Pearson. La fonction représentera aussi graphiquement les données et la droite de régression sur le même graphique. Elle représentera aussi l'histogramme des erreurs $\hat{\epsilon}_k$ dans un autre graphique.

EXERCICE 4.6 Télécharger le fichier se trouvant sur
`http ://www.cmi.univ-mrs.fr/~torresan/Matlab10/Data/brains.txt`
et l'importer dans MATLAB en utilisant soit l'instruction `importdata` (se renseigner), soit l'utilitaire d'importation se trouvant dans l'onglet `File` du menu de MATLAB . Ce fichier contient des données relatives au poids corporel et poids du cerveau dans un certain nombre d'espèces animales.

En traçant l'une contre l'autre, puis le logarithme de l'une contre le logarithme de l'autre, estimer les paramètres de ce qui vous semble être un bon modèle reliant ces deux quantités.

Remarque : La fonction `importdata` génère des *structures*, c'est à dire des variables possédant plusieurs champs. Par exemple,

```
>> tmp = importdata('fic.txt');
```

retourne une variable `tmp` possédant en général trois champs : `tmp.data` qui contient les données elles mêmes, mais aussi `tmp.textdata` et `tmp.colheaders`. On pourra examiner les contenus de ces champs pour se familiariser avec cette fonction.

4.2.2 Régression linéaire multiple

On considère maintenant le problème multidimensionnel : on suppose données des observations scalaires, mais des variables vectorielles (X_k, y_k), et on cherche à modéliser ces données sous la forme

$$y_k = X_k\beta + \alpha + \epsilon_k$$

Ici, chaque X_k est un vecteur de dimension N, que l'on va écrire en ligne, et β est un vecteur (colonne) à N lignes. A partir de là on peut former une matrice $X \in \mathcal{M}(K, N)$, et écrire le problème sous la forme $y = X\beta + \alpha + \epsilon$, ou même

$$Y = Z\gamma + \epsilon \ ,$$

en ayant posé

$$Z = \begin{pmatrix} X_{11} & X_{12} & \dots & X_{1N} & 1 \\ X_{21} & X_{22} & \dots & X_{2N} & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ X_{K1} & X_{K2} & \dots & X_{KN} & 1 \end{pmatrix} \ , \qquad \gamma = \begin{pmatrix} \beta_1 \\ \beta_2 \\ \vdots \\ \beta_N \\ \alpha \end{pmatrix}$$

De là, la solution des moindres carrés au problème de régression multiple s'obtient en résolvant

$$\min_\gamma \|Y - Z\gamma\|^2 \ ,$$

problème dont on a vu que la solution est donnée par la pseudo-inverse $Z^\dagger$ de Z :

$$\hat{\gamma} = Z^\dagger Y \ .$$

EXERCICE 4.7 Programmer une fonction `regrlinmult.m` prenant en entrée une matrice de données X et un vecteur d'observations Y, vérifiant leurs tailles respectives, et retournant le vecteur β et le scalaire α de la régression, ainsi que l'écart-type de l'erreur de régression. La fonction tracera les valeurs prédites par le modèle en fonction des vraies valeurs, l'erreur de prédiction en fonction des valeurs, et affichera l'histogramme des erreurs de régression.

EXERCICE 4.8 Télécharger le fichier se trouvant sur
`http ://www.cmi.univ-mrs.fr/~torresan/Matlab10/Data/Neige.dat`
et l'importer dans MATLAB en utilisant soit l'instruction `importdata` (se renseigner), soit l'utilitaire d'importation se trouvant dans l'onglet `File` du menu de MATLAB . Ce fichier contient des données relatives à la hauteur de neige en montagne en fonction de divers paramètres environnementaux (pente, altitude, rugosité,...).

Après avoir effectué une régression linéaire de la hauteur en fonction de quelques unes des variables, effectuer une régression linéaire multiple. Dans tous les cas, on conservera les valeurs de l'écart-type de l'erreur de régression, et on comparera les résultats obtenus.

4.2.3 Régression polynômiale

La fonction `polyfit` que nous avons déjà rencontrée plus haut effectue la régression polynômiale : étant donnée une famille d'observations $\{(x_k, y_k), k = 0, \dots K\}$, et un ordre fixé N, `polyfit` produit le polynôme P de degré N $x \to P(x)$ solution du problème de régression

$$\min_{P \in \mathcal{P}_N(\mathbb{R})} \left(\sum_{k=0}^{K} (y_k - P(x_k))^2 \right)$$

On a déjà vu que lorsque $N = K$, la régression par moindres carrés coïncide avec l'interpolation polynômiale, qui est malheureusement souvent instable. Dans le cadre de problèmes de régression, on se place souvent dans le cas $N \ll K$.

EXERCICE 4.9 Programmer une fonction `regrpolyn.m` prenant en entrée deux vecteurs x et y, un degré de polynôme N, faisant les vérifications nécessaires, et effectuant la régression polynômiale (en utilisant `polyval`). En outre, la fonction affichera dans une fenêtre graphique les données ainsi que le polynôme obtenu, et dans une autre fenêtre l'histogramme des erreurs de régression. La fonction affichera à l'écran l'écart-type de l'erreur.

EXERCICE 4.10 En utilisant les données `brain.txt`, utiliser la fonction `regrpolyn` pour effectuer une régression polynômiale de différents degrés. On tracera l'évolution de l'erreur en fonction du degré du polynôme, et on interprétera les résultats obtenus.

4.3 Devoir

Finaliser les exercices de ce chapitre. Les devoirs sont à rendre (par binôme) sous forme d'une archive contenant
- Les fichier MATLAB des fonctions développées ; celles ci devront être suffisamment commentées pour qu'un utilisateur puisse les utiliser sans effort.
- Un bref compte-rendu, décrivant dans chaque cas l'approche suivie pour résoudre le problème posé, la syntaxe de la fonction programmée si nécessaire ; le compte-rendu pourra également inclure des illustrations (sauvegardes de figures au format JPEG (extention `.jpg`) par exemple) si vous le jugez utile.
- En cas de difficultés pour inclure des figures dans le texte, on pourra plus simplement ajouter à l'archive les fichiers `.jpg`... et préciser dans le compte-rendu quelle figure on doit regarder.

5 Nombres pseudo-aléatoires, simulation

Un *générateur de nombres pseudo-aléatoires* est un algorithme qui génère une séquence de nombres présentant certaines propriétés du hasard. Par exemple, les nombres sont supposés être approximativement indépendants, et il est potentiellement difficile de repérer des groupes de nombres qui suivent une certaine règle (comportements de groupe).
Cependant, les sorties d'un tel générateur ne sont évidemment pas aléatoires ; elles s'approchent seulement des propriétés idéales des sources complètement aléatoires[1]. De vrais nombres aléatoires peuvent être produits avec du matériel qui tire parti de certaines propriétés physiques stochastiques (bruit d'une résistance par exemple).
La raison pour laquelle on se contente d'un rendu pseudo-aléatoire est :

- d'une part, il est difficile d'obtenir de "vrais" nombres aléatoires et, dans certaines situations, il est possible d'utiliser des nombres pseudo-aléatoires, en lieu et place de vrais nombres aléatoires.
- d'autre part, ce sont des générateurs particulièrement adaptés à une implémentation informatique, donc plus facilement et plus efficacement utilisables.

Les méthodes pseudo-aléatoires sont souvent employées sur des ordinateurs, dans diverses tâches comme la méthode de Monte-Carlo, la simulation ou les applications cryptographiques. Une analyse mathématique rigoureuse est nécessaire pour déterminer le degré d'aléa d'un générateur pseudo-aléatoire[2].
La plupart des algorithmes pseudo-aléatoires essaient de produire des sorties qui sont uniformément distribuées. Une classe très répandue de générateurs utilise une congruence linéaire. D'autres s'inspirent de la suite de Fibonacci en additionnant deux valeurs précédentes ou font appel à des registres à décalage dans lesquels le résultat précédent est injecté après une transformation intermédiaire.

5.1 Rappels : histogramme

Rappelons tout d'abord la définition probabiliste de l'histogramme.

DÉFINITION 5.1 *Soient* $(X_1, \dots X_N)$ *des variables aléatoires* i.i.d. *(indépendantes identiquement distribuées) à valeurs dans un ensemble* E. *Soit* $E = U_1 \cup U_2 \cup \cdots \cup U_K$ *une partition de* E *en* K *sous-ensembles. L'histogramme correspondant est la collection de variables aléatoires* $(H_1, \dots H_K)$ *définies par*

$$H_k = \sum_{n=1}^{N} \mathbb{I}_{U_k}(X_n) \ .$$

Il existe des résultats mathématiques précisant les propriétés de l'histogramme. Par exemple, dans la limite des grandes dimensions, on a le résultat de convergence suivant :

LEMME 5.1 *Avec les notations précédentes, on a*

$$\lim_{N\to\infty} H_k = \mu(U_k) \ .$$

En statistiques, un histogramme est un graphe permettant de représenter la répartition d'une variable continue. L'histogramme est un moyen simple et rapide pour représenter la distribution de cette variable. Nous utiliserons l'histogramme pour visualiser la distribution d'une collection de nombres (pseudo) aléatoires (voir par exemple la Figure 5.1. En pratique, il est important de savoir régler le nombre

[1] John von Neumann insista sur ce fait avec la remarque suivante : "Quiconque considère des méthodes arithmétiques pour produire des nombres aléatoires est, bien sûr, en train de commettre un péché".

[2] Robert R. Coveyou, du Oak Ridge National Laboratory écrivit dans un article que "la génération de nombres aléatoires est trop importante pour être confiée au hasard".

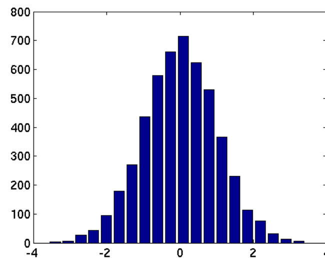

FIG. 5.1 – Histogramme

K de classes à utiliser. K ne doit être ni trop grand (auquel cas l'histogramme n'est pas représentatif, ls classes ayant souvent des effectifs trop faibles), ni trop petit (auquel cas l'histogramme ne représente pas la distribution étudiée de façon assez fine). Il existe des règles empiriques reliant le nombre de classes à la taille de l'échantillon, par exemple

$$K \approx \sqrt{N} , \quad \text{ou} \quad K \approx 1 + \frac{10\log(N)}{3} .$$

Ceci dit, l'histogramme étant avant tout un outil de visualisation, il est conseillé de l'utiliser pour plusieurs valeurs de N.

Comme on l'a déjà vu, la commande MATLAB `hist` permet de tracer l'histogramme d'une population de nombres. Elle permet également de retourner les effectifs des classes créées, ainsi que les valeurs correspondantes de la variable. Par exemple, la suite d'instructions

```
>> X = randn(5000,1) ;
>> [NN,XX] = hist(X,20) ;
```

effectue une partition du domaines de valeurs des nombres créés en 20 intervalles, et compte les effectifs de ces 20 classes, mais ne trace pas l'histogramme. Celui-ci peut être obtenu par

```
>> bar(XX,NN) ;
```

5.2 Générer des nombres uniformément distribués

5.2.1 Générateurs congruentiels

Les générateurs congruentiels génèrent des nombres entiers positifs inférieurs à une certaine valeur maximale N, en utilisant une congruence modulo N. Les suites de nombres ainsi obtenues sont évidemment périodiques avec une période inférieure ou égale à N. Les plus simples sont les générateurs de Lehmer, introduits en 1948

DÉFINITION 5.2 *Un générateur congruentiel linéaire est défini par une initialisation X_0 et une relation de congruence de la forme*

$$X_{n+1} = (aX_n + b)\,[\text{mod} N] ,$$

où a et b sont deux entiers positifs, et N est un entier positif égal à la plus grande valeur souhaitée.

La période est relativement courte, de l'ordre de la base N choisie. Les valeurs par défaut sont $a = 16807$, et $N = 2^{31} - 1$.

Il existe bien d'autres exemples, notamment

- Les générateurs de Fibonacci, définis par la récurrence

$$X_n = (X_{n-1} + X_{n-2})[\text{mod} N] ,$$

  avec une initialisation adaptée, c'est à dire la donnée de X_1 et X_2. Il faut que ceux ci soient suffisamment grands pour que les suites de nombres ainsi générées soient de bonne qualité.
- Les générateurs de Fibonacci généralisés, définis par

$$X_{n+1} = (X_{n-k} + X_{n-\ell+c})[\text{mod} N] ,$$

  avec c fixé, et une initialisation adaptée, c'est à dire la donnée de $X_1, X_2, \dots X_k$ (si on suppose $k > \ell$).

– Générateurs non-linéaires : par exemple

$$X_{n+1} = X_n(X_n + 1)[\text{mod}2^k] ,$$

pour un k assez grand, et une initialisation bien choisie $X_2 = 2[\text{mod}4]$

EXERCICE 5.1
1. Ecrire une fonction `fibogen`, qui génère des nombres pseudo-aléatoires uniformément distribués entre 0 et 1 avec l'algorithme de Fibonacci. La fonction prendra en entrée la base N, les valeurs initiales X_1 et X_2, ainsi que le nombre de valeurs à générer.
2. Ecrire une fonction `lincongen`, qui génère des nombres pseudo-aléatoires uniformément distribués entre 0 et 1 avec l'algorithme congruentiel linéaire ci-dessus. La fonction prendra en entrée la base N, les entiers a et b, la valeurs initiale X_1, ainsi que le nombre de valeurs à générer.
3. Dans les deux cas, étudier la qualité du générateur obtenu. On pourra par exemple calculer un histogramme, et calculer l'erreur de celui-ci par rapport à la distribution attendue, et étudier l'évolution de cette erreur en fonction du nombre N de nombres aléatoires générés.

Dans cet exercice, il sera utile d'effectuer des tests avec les générateurs ainsi construits, pour différents choix de paramètres, et d'interpréter les résultats obtenus.

5.2.2 Les générateurs dans MATLAB

Comme on l'a vu, l'instruction `rand` permet de générer des séquences de nombres pseudo-aléatoires aussi "indépendants" que possible, et distribués selon une loi proche de la loi $\mathcal{U}([0,1])$. Les paramètres sont optimisés pour fournir des nombres d'aussi bonne qualité que possible.
`rand` propose trois méthodes, correspondant à trois algorithmes différents, que l'on peut choisir en utilisant l'instruction

```
>> rand(methode,etat)
```

La variable `etat` est un entier positif caractérisant l'état courant du système (en gros, l'initialisation du générateur), et la variable `methode` est l'un des trois choix suivants
- `'state'` : le générateur par défaut, dont la période peut théoriquement atteindre 2^{1492}.
- `'seed'` : l'ancien générateur par défaut, basé sur un gégérateur congruentiel multiplicatif, et dont la période vaut $2^{31} - 2$.
- `'twister'` : utilise le twister de Mersenne, l'un des meilleurs générateurs actuels (plus complexe, et donc aussi plus lent que les autres), dont la période vaut $(2^{19937} - 1)/2$.

L'instruction

```
>> s = rand(methode) ;
```

retourne une chaîne de caractères contenant l'état du système pour la méthode choisie.

Une fois la méthode et l'état initial choisis, on utilise le générateur comme on l'a déjà vu :

```
>> X = rand(m,n) ;
```

génère une matrice à m lignes et n colonnes contenant des nombres pseudo-aléatoires avec une distribution aussi proche que possible de celle d'un vecteur i.i.d. $\mathcal{U}([0,1])$.
`randn` génére des matrices pseudo-aléatoires approchant une loi i.i.d. $\mathcal{N}(0,1)$. L'instruction

```
>> X = s*randn(M,N) + m ;
```

où m et s sont des réels, génère $M \times N$ nombres pseudo-aléatoires i.i.d. suivant approximativement une loi $\mathcal{N}(m, s^2)$. Pour `randn`, seuls deux méthodes sont disponibles : `'seed'` et `'state'`.

On pourra se documenter aussi sur `randperm` (permutations aléatoires).

5.3 Simuler d'autres distributions

Les méthodes ci-dessus permettent de simuler des distributions uniformes d'entiers entre 0 et $N-1$ (où N est très grand), et donc par division par N de réels (plutôt des rationnels) dans $[0,1]$. On va maintenant voir comment utiliser ces derniers pour simuler d'autres distributions.

5.3.1 La toolbox Stixbox

Pourquoi se fatiguer quand quelqu'un d'autre a déjà fait le travail ? La Stixbox, disponible sur
`http ://www.maths.lth.se/matstat/stixbox/stixbox.tar`
contient des implémentations de la plupart des densités, des fonctions de répartition, des fonctions quantiles, et des générateurs de nombres aléatoires des lois classiques. La plupart sont basés sur l'inversion de la fonction de répartition, sauf quelques uns qui utilisent le rejet.

Prenons l'exemple de la distribution χ^2. Une variable aléatoire X suit une loi χ^2 à k degrés de liberté si la densité de X notée f_X est :

$$f_X(t) = \frac{1}{2^{\frac{k}{2}}\Gamma(\frac{k}{2})} t^{\frac{k}{2}-1} e^{-\frac{t}{2}} \, \forall t \in \mathbb{R}^+ ,$$

où Γ est la fonction gamma d'Euler.
Stixbox implémente quatre fonctions reliées à la distribution χ^2 :
- `rchisq(n,k)` : génére n nombres pseudo-aléatoires distribués suivant une loi χ^2 à k degrés de liberté.
- `dchisq(x,k)` : évalue la densité de probabilités d'une variable χ^2 à k degrés de liberté au point x.
- `pchisq(x,k)` : évalue la fonction de répartition d'une variable χ^2 à k degrés de liberté au point x.
- `qchisq(x,k)` : évalue la réciproque de la fonction de répartition d'une variable χ^2 à k degrés de liberté au point x.

La suite d'instructions suivante permet de générer la figure 5.2.

```
>> y = rchisq(10000,k) ;
>> subplot(2,1,1) ;
>> hist(y,50) ;
>> x = linspace(0,20,1000) ;
>> subplot(2,1,2) ;
>> plot(x,dchisq(x,k)) ;
```

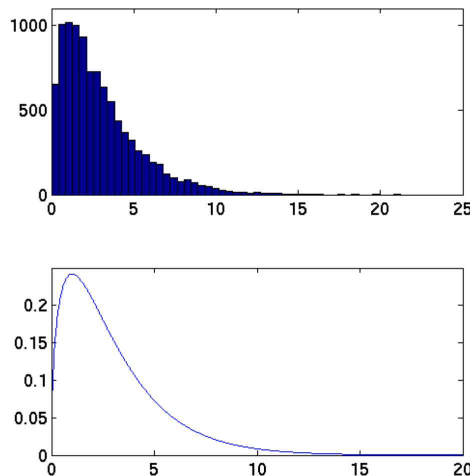

Fig. 5.2 – Densité de la distribution χ^2 à 3 degrés de liberté (bas), et histogramme correspondant (sur 10 000 réalisations, en haut)

On peut également tracer la fonction de répartition et sa réciproque (voir figure 5.3) via les instructions

```
>> x = linspace(0,20,1000) ;
>> subplot(2,1,1) ;
>> plot(x,pchisq(x,k)) ;
>> subplot(2,1,2) ;
>> x = linspace(0,1,100) ;
>> plot(x,qchisq(x,k)) ;
```

La syntaxe est similaires pour un certain nombre d'autres distributions classiques (Fisher, Gamma, Gumbel,...).

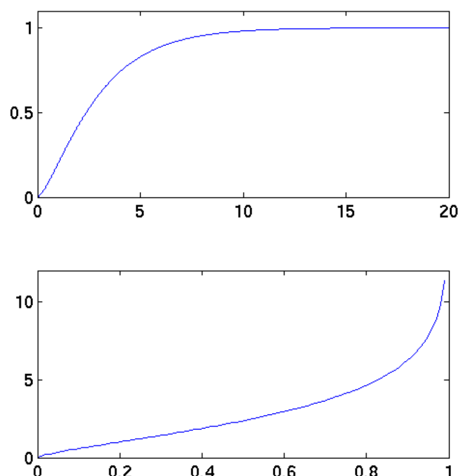

Fig. 5.3 – Fonction de répartition d'une loi de χ^2 à 3 degrés de liberté (haut), et sa réciproque (bas)

5.3.2 Méthode de la fonction de répartition

La premi'ere méthode est basée sur la fonction de répartition de la variable aléatoire que l'on désire simuler, et exploite le résultat suivant :

Lemme 5.2 *Soit F la fonction de répartition d'une variable aléatoire sur $\mathbb{R}$, et soit $U \sim \mathcal{U}([0,1])$ une variable aléatoire uniforme sur $[0,1]$. Alors la variable aléatoire $V = F^{-1}(U)$ admet F pour fonction de répartition.*

Cette méthode est facile à mettre en oeuvre lorsque la réciproque de la fonction de répartition est connue. Si tel n'est pas le cas, on peut toutefois se baser sur une approximation de celle-ci.

Remarque : Il est possible de réinterpréter la fonction `VAdiscrete` précédemment développée en termes de cette approche. Supposons en effet que nous voulions simuler une variable aléatoire discrète prenant les valeurs $x_1 \leq x_2 \leq \cdots \leq x_N$, avec probabilités $p_k, k = 1, \ldots N$. La fonction de répartition correspondante est comme indiqué en figure 5.4, où on pose

$$F_0 = 0 \ , \quad F_1 = p_1 \ , \quad F_2 = p_1 + p_2, \ldots \quad F_k = \sum_{\ell=1}^{k} p_\ell \ , \ldots F_N = 1 \ .$$

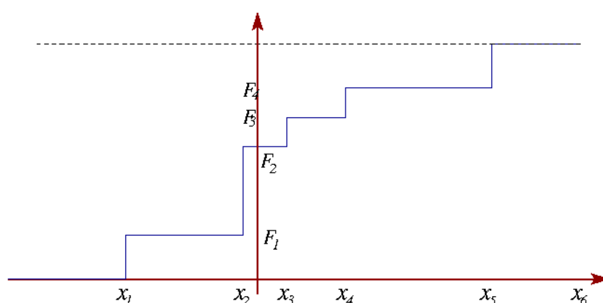

Fig. 5.4 – Fonction de répartition d'une variable aléatoire discrète

La fonction de répartition est constante par morceaux, et n'est pas inversible à proprement parler. On convient donc d'associer à tout $f \in [0,1]$ la valeur $X(f) = x_k$ telle que $x_{k-1} \leq f < x_k$.
Il est facile de vérifier qu'on obtient bien de la sorte la distributiuon désirée.

EXERCICE 5.2 Ecrire une fonction `monranexp.m` générant des réalisations d'une variable aléatoire exponentielle de paramètre λ, de densité

$$\rho_\lambda(x) = \lambda e^{-\lambda x} \ , \quad x \in \mathbb{R}^+ \ .$$

On explicitera tout d'abord la fonction de répartition, puis on utilisera celle-ci. Les variables d'entrée de la fonction seront λ et le nombre de nombres aléatoires à générer.

EXERCICE 5.3 Ecrire une fonction générant des réalisations d'une variable aléatoire de Cauchy, de densité

$$\rho(x) = \frac{1}{\pi}\frac{1}{1+x^2} \ , \quad x \in \mathbb{R} \ .$$

On explicitera tout d'abord la fonction de répartition, puis on utilisera celle-ci. La variable d'entrée de la fonction sera le nombre de nombres aléatoires à générer.

EXERCICE 5.4 Ecrire une fonction `monrandn1.m` générant un nombre donné de réalisations d'une variable aléatoire Gaussienne de moyenne a et écart-type s donnés, en utilisant la fonction de répartition. On se documentera sur les fonctions `erf` et `erfinv` de MATLAB .

5.3.3 Méthode de Box-Müller pour des Gaussiennes

Dans le cas de variables Gaussiennes (pour lesquelles la réciproque de la fonction de répartition n'a pas d'expression explicite), il existe une autre méthode, basée sur le résultat suivant :

LEMME 5.3 *Soient $U, V \sim \mathcal{U}([0,1])$ deux variables aléatoires indépendantes. Soient X et Y définies par*

$$\begin{cases} X &= \sqrt{-2\ln(U)}\sin(2\pi V) \\ Y &= \sqrt{-2\ln(U)}\cos(2\pi V) \end{cases}$$

Alors X et Y sont deux variables aléatoires normales centrées réduites (c'est à dire $X, Y \sim \mathcal{N}(0,1)$) indépendantes.

Cette transformation vient du fait que, dans un système Cartésien à deux dimensions où les coordonnées X et Y sont deux variables aléatoires indépendantes normales, les variables aléatoires R^2 et Θ sont elles aussi indépendantes et peuvent s'écrire

$$R^2 = -2\ln U \qquad \Theta = 2\pi V \ .$$

EXERCICE 5.5 Ecrire une fonction `monrandn2.m` générant un nombre donné de réalisations d'une variable aléatoire Gaussienne de moyenne a et écart-type s donnés, en utilisant l'algorithme de Box-Müller.

5.3.4 Méthodes de rejet

Le principe des méthodes de rejet est d'utiliser une variable aléatoire que l'on sait simuler pour simuler une autre variable aléatoire plus complexe, en sélectionnant les "bonnes réalisations" de la première. Plus précisément, on se base sur le résultat suivant

PROPOSITION 5.1 *Soient ρ et ρ_0 deux densités de probabilités telles qu'il existe une constante positive K vérifiant*

$$\rho(x) \leq K\rho_0(x) \quad \forall x \ .$$

Soient X une variable aléatoire de densité ρ_0, et soit $U \sim \mathcal{U}([0,1])$ une variable aléatoire uniformément distribuée entre 0 et 1, indépendante de X.

Soit E l'évènement

$$E = \{KU\rho_0(X) < \rho(X)\} \tag{$\sharp$}$$

Alors, la loi conditionnelle de X sachant E a pour densité ρ.

De là on tire un algorithme simple pour simuler des réalisations d'une variable aléatoire de densité ρ :
– Tirer au hasard X et U suivant la densité ρ_0 et la loi $\mathcal{U}(0,1)$ respectivement.
– Calculer $KU\rho_0(X)$ et $\rho_0(X)$.
– Si la condition ($\sharp$) est remplie, conserver la valeur obtenue.
Prenons l'exemple d'une distribution de Cauchy vue précédemment

$$\rho_0(x) = \frac{1}{\pi}\frac{1}{1+x^2} \ , \quad x \in \mathbb{R} \ ,$$

et d'une Gaussienne centrée réduite de densité

$$\rho(x) = \frac{1}{\sqrt{2\pi}} \exp\left(-x^2/2\right)$$

EXERCICE 5.6 Montrer que

$$\rho(x) \leq \sqrt{\frac{2\pi}{e}}\rho_0(x) \ .$$

Utiliser cette remarque pour construire un générateur de nombres pseudo-aléatoires Gaussiens (centrés réduits) à partir d'un générateur de distribution de Cauchy (le votre, ou celui de la STIXBOX), et de la fonction MATLAB `rand`.

Notons que les méthodes que nous avons utilisées au premier chapitre pour simuler des distributions aléatoires dans un disque on une sphère sont des versions simplifiées de la méthode du rejet. On va maintenant en voir des versions plus efficaces.

EXERCICE 5.7 Utiliser la méthode du rejet pour simuler un vecteur aléatoire uniformément distribué dans un disque de rayon R_0 et de centre (a,b) du plan, à partir de la fonction `rand`. On pourra commencer par le cas $a = b = 0$ et $R_0 = 1$, et on utilisera le fait que pour un tel vecteur aléatoire, en passant en coordonnées polaires, les variables radiale R et angulaire Θ sont indépendantes, et distribuées respectivement avec une densité $\rho_R(r) = 2r, r \in [0,1]$ et selon une loi uniforme $\mathcal{U}(0,2\pi)$.

5.3.5 Gaussiennes multivariées

Passons maintenant au cas multivarié. On rappelle qu'étant donnée une matrice $\mathcal{C} \in \mathcal{M}(N,N)$ symétrique définie positive, et un vecteur $b \in \mathbb{R}^N$, la densité de probabilités Gaussienne multivariée associée est la fonction de N variables définie par

$$\rho(x) = \frac{1}{(2\pi)^{N/2}\det(\mathcal{C})} \exp\left\{\langle (x-b), \mathcal{C}^{-1}(x-b)\rangle\right\} \ ,$$

où on a noté $\langle\cdot,\cdot\rangle$ le produit scalaire dans $\mathbb{R}^N$. b est la moyenne, et $\mathcal{C}$ est la matrice de variance-covariance : si $X = (X_1, \dots X_N)$ est un vecteur aléatoire associé à cette distribution, on a

$$b_k = \mathbb{E}\{X_k\} \ , \quad \mathcal{C}_{k\ell} = \mathbb{E}\{(X_k - b_k)(X_\ell - b_\ell)\} \ , \quad k,\ell = 1, \dots N \ .$$

Pour simuler de tels vecteurs aléatoires Gaussiens, on se base sur la décomposition de Cholesky de la matrice de Variance-Covariance.

Lemme 5.4 (Factorisation de Cholesky) *Soit A une matrice symétrique définie positive, il existe au moins une matrice réelle triangulaire inférieure L telle que :*

$$A = LL^T$$

Remarque : On peut également imposer que les éléments diagonaux de la matrice L soient tous positifs, et la factorisation correspondante est alors unique.

Supposons maintenant que $W = \{W_1, \dots W_N\}$ soit un vecteur aléatoire Gaussien centré réduit : ses composantes W_k sont des variables aléatoires centrées réduites dindépendantes. Soit $\mathcal{C} = LL^T$ une décomposition de Cholesky de la matrice définie positive $\mathcal{C} \in \mathcal{M}(N, N)$, et posons $X = LW$. Alors

$$\mathbb{E}\{X_k X_\ell\} = \sum_{m,n} L_{km} L_{\ell n} \mathbb{E}\{W_m W_n\} = \sum_m L_{km} L_{\ell m} = LL^T_{k\ell} = \mathcal{C}_{k\ell}\ .$$

Des combinaisons linéaires de variables aléatoires Gaussiennes étant toujours Gaussiennes, on a donc montré

Lemme 5.5 *Soit $W = \{W_1, \dots W_N\}$ soit un vecteur aléatoire Gaussien centré réduit, soit $B \in \mathbb{R}^N$ un vecteur, soit $\mathcal{C} \in \mathcal{M}(N, N)$ une matrice symétrique définie positive, soit $\mathcal{C} = LL^T$ une décomposition de Cholesky de $\mathcal{C}$. Alors le vecteur aléatoire défini par*

$$X = LW + B$$

est un vecteur aléatoire Gaussien centré, de matrice de covariance $\mathcal{C}$.

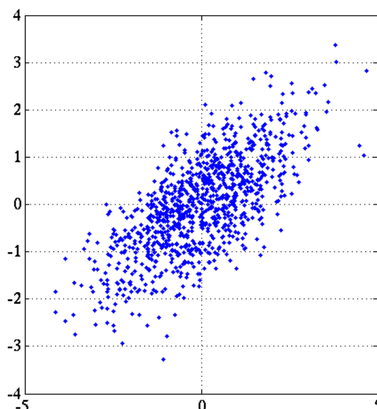

Fig. 5.5 – 2000 réalisations d'un vecteur aléatoire Gaussien dans le plan

Exercice 5.8 Ecrire une fonction qui, partant d'une matrice de variance-covariance donnée $C \in \mathcal{M}(N, N)$ (symétrique et définie positive), engendre M réalisations d'un vecteur aléatoire Gaussien centré, de matrice de variance-covariance C.

Attention : se document sur `chol` !

Pour visualiser les résultats obtenus, on pourra par exemple se placer dans le cas bidimensionnel, et tester avec une matrice diagonale, ou la matrice

$$C = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}$$

et tracer dans le plan les points obtenus. On pourra faire de même en trois dimensions.

Un exemple de visualisation se trouve en Figure 5.5 ; on peut y remarquer la forme "ellipsoïdale" du nuage de points ainsi généré.

5.4 Une application : intégration de Monte-Carlo

Il peut arriver que l'on ait à évaluer numériquement des intégrales difficilement calculables par des méthodes d'intégration déterministes (par exemple, en grande dimension). On peut alors avoir recours à des méthodes d'intégration utilisant des nombres pseudo-aléatoires.
Par exemple, supposons que l'on ait à calculer une intégrale d-dimensionnelle de la forme

$$I = \int_{\mathcal{D}} g(\underline{x}) f(\underline{x})\, d\underline{x} \ ,$$

$\mathcal{D} \subset \mathbb{R}^d$ est un domaine, et f est telle $\int_{\mathcal{D}} f(\underline{x})\, d\underline{x} = 1$. f peut alors être interprétée comme une densité de probabilités, et l'intégrale comme une espérance mathématique. Si on sait générer des vecteurs pseudo-aléatoires suivant la loi de f, on peut alors approximer l'intégrale sous la forme

$$\hat{I}_N = \frac{1}{N} \sum_{n=1}^{n} g(X_n) \ ,$$

où les X_n sont N vecteurs pseudo-aléatoires indépendants suivant f.
La loi des grands nombres

THÉORÈME 5.1 *Soient $X_1, \ldots X_N$ N vecteurs aléatoires indépendants de même distribution de densité $f(\underline{x})$ telle que $\mathbb{E}\{g(X_n)\} = \mu$ et $\mathbb{E}\left\{(g(X_n))^2\right\} < \infty$. Soit*

$$g_N = \frac{1}{N} \sum_{n=1}^{N} g(X_n) \ .$$

Alors pour tout $\epsilon > 0$,

$$\lim_{N \to \infty} \mathbb{P}\{|g_N - \mu| > \epsilon\} = 0 \ .$$

Un grand avantage de cette méthode est que quelle que soit la dimension d, l'erreur $|g_N - \mu|$ diminue comme $1/\sqrt{N}$ quand N augmente. Par contre l'erreur est proportionnelle à $\mathrm{var}\{g(X_n)\}$ ... et il faut simuler Nd variables aléatoires pour obtenir N vecteurs aléatoires de dimension d.