الجمهورية الجزائرية الديمقراطية الشعبية
République algérienne démocratique et populaire
وزارة التعليم العالي والبحث العلمي
Ministère de l'Enseignement supérieur et de la Recherche scientifique

Université Abderrahmane MIRA- Bejaia
Faculté de Technologie
Département de Génie Civil
Année universitaire 2024/2025

جامعة عبد الرحمان ميرة – بجاية
كلية التكنولوجيا
قسم الهندسة المدنية
السنة الجامعية 2024/2025

Polycopié de :

Travaux pratiques de méthodes numériques

Préparé par: Abderrezak Bouziane

Maître de conférence - Département de génie civil -

Faculté de technologie - Université de Bejaia

Public cible : étudiants en 2ième année licence genie civil

Année universitaire 2024/2025

Préface

Ce polycopié de travaux pratiques est destiné aux étudiants de **deuxième année licence** en **Génie Civil**, dans le cadre de l'enseignement des **méthodes numériques** appliquées à l'ingénierie. Il a pour objectif de fournir une approche pratique et progressive des principales techniques numériques utilisées pour résoudre les problèmes courants rencontrés dans le domaine.

L'ensemble des travaux pratiques présentés dans ce document s'appuie exclusivement sur le logiciel **MatLab**, outil largement utilisé dans le milieu académique et professionnel pour le calcul scientifique, l'analyse de données et la simulation numérique. À travers une série d'exercices guidés, les étudiants apprendront à mettre en œuvre des méthodes classiques telles que la résolution d'équations non linéaires, l'interpolation, l'intégration numérique, la résolution d'équations différentielles, ainsi que les systèmes d'équations linéaires.

Une introduction à l'algorithmique, ainsi qu'une prise en main progressive de l'environnement MatLab, permettent d'acquérir les bases nécessaires à la compréhension et à l'implémentation des méthodes étudiées. Les chapitres sont organisés de manière à favoriser l'apprentissage autonome et à renforcer le lien entre la théorie mathématique et son application concrète en ingénierie.

Ce support pédagogique a été conçu pour accompagner les étudiants dans le développement de compétences numériques essentielles, en leur donnant les outils nécessaires pour analyser, modéliser et résoudre efficacement des problèmes d'ingénierie à l'aide de **MatLab**.

Abderrezak Bouziane

Année universitaire 2024/2025

Semestre : S4
Unité d'enseignement: UEM 2.2
Matière 4 : TP Méthodes Numériques
VHS: 22h30 (TP : 1h30)
Crédits: 2
Coefficient: 1

Objectifs de l'enseignement:

Programmation des différentes méthodes numériques en vue de leurs applications dans le domaine des calculs mathématiques en utilisant un langage de programmation scientifique.

Connaissances préalables recommandées:

Méthode numérique, Informatique 2 et informatique 3.

Contenu de la matière :

Chapitre 1 : Résolution d'équations non linéaires (3 semaines)
1.Méthode de la bissection. 2. Méthode des points fixes, 3. Méthode de Newton-Raphson

Chapitre 2 : Interpolation et approximation (3 semaines)
1.Interpolation de Newton, 2. Approximation de Tchebychev

Chapitre 3 : Intégrations numériques (3 semaines)
1.Méthode de Rectangle, 2. Méthode de Trapezes, 3. Méthode de Simpson

Chapitre 4 : Equations différentielles (2 semaines)
1.Méthode d'Euler, 2. Méthodes de Runge-Kutta

Chapitre 5 : Systèmes d'équations linéaires (4 semaines)
1.Méthode de Gauss- Jordon, 2. Décomposition de Crout et factorisation LU, 3. Méthode de Jacobi, 4. Méthode de Gauss-Seidel

Mode d'évaluation :

Contrôle continu : 100 %.

Références :

1. Algorithmique et calcul numérique : travaux pratiques résolus et programmation avec les logiciels Scilab et Python / José Ouin, . - Paris : Ellipses, 2013 . - 189 p.
2. Mathématiques avec Scilab : guide de calcul programmation représentations graphiques ; conforme au nouveau programme MPSI / Bouchaib Radi, ; Abdelkhalak El Hami . - Paris : Ellipses, 2015 . - 180 p.
3. Méthodes numériques appliquées : pour le scientifique et l'ingénieur / Jean-Philippe Grivet, - Paris : EDP sciences, 2009 . - 371 p.

Table des matières

Introduction

Dans de nombreux domaines scientifiques et techniques, les ingénieurs et chercheurs sont confrontés à des problèmes mathématiques complexes qui ne peuvent être résolus analytiquement. Les **méthodes numériques** offrent des outils puissants permettant d'approcher ces solutions de manière efficace et précise.

Ce cours vise à introduire les concepts fondamentaux du calcul numérique et à familiariser les étudiants avec les techniques utilisées pour résoudre différents types de problèmes, tels que la recherche de racines d'équations, l'interpolation, l'intégration numérique, la résolution d'équations différentielles et la résolution de systèmes linéaires.

L'apprentissage s'appuiera sur la **programmation scientifique** avec le logiciel **MATLAB**, qui permettra d'implémenter, de tester et d'analyser les algorithmes étudiés. L'objectif est de comprendre non seulement le fonctionnement de ces méthodes, mais aussi leurs performances, leurs erreurs et leurs domaines d'application en ingénierie.

À travers des **travaux pratiques**, les étudiants développeront des compétences essentielles pour appliquer ces outils aux problématiques du **génie civil**, notamment en modélisation, simulation et analyse des structures et des matériaux. Ce cours constitue ainsi un socle fondamental pour aborder des études plus avancées en calcul numérique et en simulation informatique.

Organisation du cours et contenu

Le cours est structuré en cinq chapitres abordant des thèmes fondamentaux du calcul numérique.

- **Résolution numérique des équations non linéaires (3 semaines)**
- Méthode de la bissection
- Méthode des points fixes
- Méthode de Newton-Raphson

- **Interpolation et approximation (3 semaines)**
- Interpolation de Newton
- Approximation de Tchebychev

- **Intégration numérique (3 semaines)**
- Méthode des rectangles
- Méthode des trapèzes
- Méthode de Simpson

- **Résolution des équations différentielles (2 semaines)**
- Méthode d'Euler
- Méthode de Runge-Kutta

- **Systèmes d'équations linéaires (4 semaines)**
- Méthode de Gauss-Jordan
- Décomposition de Crout et factorisation LU
- Méthode de Jacobi
- Méthode de Gauss-Seidel

Méthodologie et évaluation

L'enseignement est assuré sous forme de travaux pratiques (TP) avec une durée totale de **22h30** (séances de **1h30** chacune). L'évaluation repose exclusivement sur un **contrôle continu** (**100 %**).

Rappel sur l'algorithmique [1]

1 Définition de l'algorithmique

L'algorithmique englobe l'ensemble des principes, des règles et des techniques utilisées pour définir et concevoir des algorithmes [1].

Définition d'un algorithme

Un algorithme est une suite finie d'opérations élémentaires formant un processus de calcul ou une méthode de résolution pour un problème donné.

Remarque : Le résultat doit être obtenu dans un te mps fini. Ces opérations élémentaires consistent en une série d'instructions conçues pour transformer les données initiales afin d'atteindre le résultat recherché.

Les algorithmes se trouvent également dans des situations de la vie courante. En voici deux exemples :

Algorithme d'utilisation d'un distributeur automatique de boissons chaudes :

Instruction 1 : Mettre une pièce dans le distributeur.

Instruction 2 : Appuyer sur le bouton correspondant à la boisson choisie.

Instruction 3 : Une structure répétitive

TantQue le bip sonore n'a pas été émis

Attendre la fin du remplissage du gobelet.

FinTantQue

Instruction 4 : Prendre le gobelet et consommer la boisson.

Algorithme de fabrication du pain :

Instruction 1 : Mélanger la farine, la levure et l'eau.

Instruction 2 : Pétrir 15 à 20 minutes.

Instruction 3 : Laisser lever la pâte durant 2 heures.

Instruction 4 : Mettre au four à 180 0 pendant I heure.

Instruction 5 : Une structure alternative :

Si le pain n'est pas assez doré Alors

Laisser cuire encore 10 minutes.

Sinon

Sortir le pain du four.

FindeSi

Instruction 6 : Laisser refroidir le pain sur une grille.

2 Algorithmique et programmation

2.1 Un algorithme puis un programme

Un algorithme n'est pas un programme. Il s'agit d'une solution au problème considéré, indépendante des spécificités d'un langage de programmation particulier.

Un programme, en revanche, est l'implémentation concrète d'un algorithme à l'aide d'un langage de programmation spécifique (comme Python, C++, MATLAB, Fortran, etc.). Il consiste à traduire l'algorithme en un ensemble d'instructions exécutables.

Par exemple, lors de la programmation, il est nécessaire de définir le format des variables utilisées, une problématique propre à l'implémentation qui n'existe pas au niveau de la conception algorithmique.

Le code source désigne le texte écrit dans un langage de programmation et constituant le programme.

2.2 Les éléments de base d'un algorithme

1. Identification des données nécessaires (La préparation du traitement)

Il est essentiel d'identifier les données indispensables pour résoudre le problème. Ces données peuvent être de différents types :

Numériques :

- Saisies au clavier par l'utilisateur
- Lue à partir d'un fichier contenant des nombres.

Graphiques :

- Position du pointeur de la souris.

Textuelles :

- Saisie au clavier par l'utilisateur ;
- Lecture d'un fichier contenant du texte.

Il est également crucial d'identifier les résultats intermédiaires à mémoriser pour les traitements ultérieurs. Ces données intermédiaires, indispensables à la résolution, peuvent nécessiter l'utilisation de variables auxiliaires pour préserver l'intégrité des données initiales.

2. Définition du traitement

Cela consiste à déterminer toutes les instructions nécessaires pour automatiser la résolution du problème et produire les résultats attendus. Chaque étape du traitement doit être clairement définie et structurée.

3. La sortie des résultats

Les résultats obtenus peuvent être présentés sous différentes formes :

- **Affichage** à l'écran (nombres, textes, graphiques).
- **Impression** sur papier.
- **Écriture** dans un fichier.

2.2.1 Conventions d'écriture d'un algorithme

1. Algorithme graphique ou organigramme de programmation

Un algorithme peut être représenté graphiquement à l'aide d'un organigramme. Celui-ci utilise des symboles spécifiques pour les traitements et les tests (rectangles, losanges, etc.), reliés par des lignes qui illustrent le flux des contrôles.

Il existe une norme internationale, l'ISO 5807, qui définit de manière détaillée les symboles standard à utiliser pour représenter un algorithme informatique de façon normalisée.

Exemple :

L'organigramme de la figure 1 représente une structure alter

Si la « Condition » est vérifiée **Alors**

Effectuer le « Traitement 1 »

Sinon

Effectuer le « Traitement 2 »

FinSi

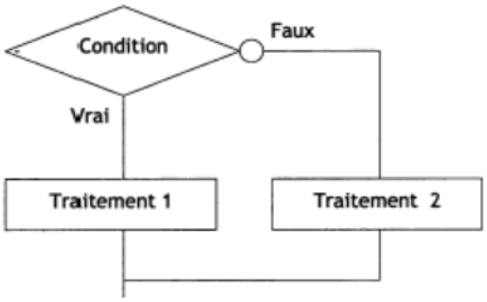

Figure 1. Structure alternative [1]

2. L'algorithme textuel ou pseudo-code

Le pseudo-code est une façon de décrire un algorithme sans référence à un langage de programmation en particulier. Il ressemble cependant à un langage de programmation authentique mais dont on aurait retiré la plupart des problèmes de syntaxe.

Le pseudo-code est susceptible de varier légèrement d'un programmeur à un autre.

Exemple :

d **prend la valeur de** b^2-4*a*c

Si d<0 **Alors**

Afficher « aucune solution. »

Sinon

Afficher « Une ou deux racines »

Finsi

3 Les instructions

Les instructions constituent la structure des algorithmes et leur assemblage, dans un ordre précis conduit au résultat attendu. Elles sont écrites en pseudo-code. Un exemple concret d'écriture en langage Matlab est également proposé.

3.1 Les instructions pour traiter les données

II s'agit d'instructions de base comme la lecture de données, l'affectation dans des variables et l'écriture de données.

3.1.1 L'affectation de données dans des variables

L'affectation permet d'attribuer une valeur à une variable désignée par son identificateur. Un identificateur est une suite de lettres et chiffres (sans espaces) qui doit être choisi pour que l'algorithme soit immédiatement lisible et interprétable.

- Affectation en pseudo-code

identificateur **prend la valeur** valeur

d **prend la valeur** 5

L'affectation remplace la valeur de la variable par la nouvelle. Ainsi l'instruction « d **prend la valeur** 5 » affecte la valeur 5 à la variable dont d est l'identificateur et ceci quelle que soit la valeur contenue au préalable dans la variable d (laquelle sera perdue).

- Affectation en langage Matlab

```
d = 5 ;
```

Le point-virgule est optionnel. Il permet de ne pas afficher la valeur de la variable d dans la fenêtre des commandes au cours du déroulement du programme.

3.1.2 La lecture (ou entrée) des données

La lecture de données peut se faire par interrogation de l'utilisateur ou par extraction à partir d'un fichier.

- Lecture des données en pseudo-code

Saisir identificateur.

Exemple

Saisir p

- Lecture des données en langage Matlab

On ne considère que le cas de la lecture de données par interrogation de l'utilisateur. La procédure de lecture de données dans un fichier dépend en effet de la disposition des données dans ce fichier et ne peut donc pas être décrite de manière générale.

w = input ("Entrer votre prénom : ")

u = input ("Entrer les bornes de 1''intervalle [a,b]=")

p = input ("Entrer la valeur de la précision")

La variable **w** est une chaîne de caractère ("string") contenant le prénom saisi.

La variable **u** est un vecteur contenant deux nombres réels.

Si l'utilisateur a saisi [3,9], alors u(l) (ou u(1,1)) est égal à 3 et u(2) (ou u(1,2)) est égal à 9.

La variable **p** contient un nombre réel.

3.1.3 L'écriture (ou sortie) des données

L'écriture des données permet d'afficher, pour l'utilisateur, les valeurs des variables après traitement.

- **Ecriture des données en pseudo-code**

Afficher identificateur

Pour écrire des informations non contenues dans une variable :

Afficher "message".

Exemple :

Afficher x1

Afficher "L'équation n'a aucune solution"

- **Ecriture des données en langage Matlab** [2]

→ La fonction **disp**

Affichage d'un message :

disp ("f doit changer de signe ")

Affichage d'une valeur numérique

disp(a)

→ La fonction **fprintf**

- ***Affichage d'un message***

fprintf ("f doit changer de signe sur 1' 'intervalle [a, b]! %s\n ")

La chaîne de caractères "%s\n " est appelée chaîne de formatage

%s indique à la fonction fprintf qu'il s'agit de l'affichage d'une chaîne de caractères.

\n spécifie un retour à la ligne suite à cet affichage.

- ***Affichage d'une valeur numérique***

fprintf ("Une racine double : x1=%f\n",x1)

%f indique à la fonction **fprintf** qu'il s'agit de l'affichage d'un nombre réel (float).

\n spécifie un retour à la ligne suite à cet affichage (retour chariot).

- ***Affichage de plusieurs variables***

fprintf ("Encadrement : %f%s%f\n",a, "< xsol < ",b)

L'affichage est le suivant : a < xsol < b

3.2 Les instructions ou structures de contrôle

Le traitement de données se fait parfois sous conditions ou de manière spécifique. On parle alors de structures de contrôle.

3.2.1 La structure alternative

Selon qu'une certaine condition est vérifiée ou non, on fera un certain traitement. Un traitement est constitué par une ou plusieurs instructions.

- ***Structure alternative en pseudo-code***

Si (condition) **alors**

(Traitement 1)

Sinon

(Traitement 2)

FinSi

Exemple

Si d<0 **alors**

Afficher "Aucune solution"

Sinon

Afficher "Une ou deux solutions"

FinSi

On peut également utiliser des structures alternatives imbriquées

Si (condition 1) **alors**

Si (condition 2) **alors**

(Traitement 1)

Sinon

(Traitement 2)

FinSi

Sinon

(Traitement 3)

FinSi

- **Exemple**

d **prend la valeur** b^2 - 4*a*c

Si d>0 **Alors**

x1 **prend la valeur** (-b-racine(d))/2a

x2 **prend la valeur** (-b + racine(d))/2a

Afficher x1 , x2 (deux racines distinctes)

SinonSi d=0 **Alors**

x1 **prend la valeur** -b/2a

Afficher x1 (une racine double)

Sinon

Afficher "Aucune solution"

FinSi

- **Structure alternative en langage Matlab**

```
d=b^2-4*a*c;
if d>0
       x1=(-b-sqrt (d) ) / (2 *a) ;
       x2=(-b+sqrt (d) ) / (2 *a) ;
       fprintf ("%s\n", "RESULTATS : ")
       fprintf ("Deux racines distinctes : x1=%f%s%f\n",x1, " et x2=",x2)
               elseif d==0
               x1=-b/(2*a);
               fprintf ("%s\n", "RESULTATS : ")
               fprintf ("Une racine double: x1=%f \n",x1)
else
fprintf ("%s\n", "RESULTATS : ")
fprintf ("%s\n", "Aucune solution !")
end
```

3.2.2 Les structures répétitives

Les structures répétitives permettent d'exécuter plusieurs fois de suite le même traitement c'est à dire la même série d'instructions.

On utilise pour cela un compteur (par exemple une variable **k**) ou une condition pour contrôler le nombre de fois que les instructions sont répétées.

Dans le cas d'un compteur, pour chaque répétition, la structure répétitive incrémente ou décrémente la valeur de la variable **k** d'une valeur prédéfinie.

Dans le cas d'une structure répétitive avec une condition, la structure vérifie si la condition est toujours vérifiée avant chaque répétition.

3.2.2.1 Structures répétitives en pseudo-code

Structure répétitive avec un compteur (structure répétitive de 1 à n)

Pour k **de** 1 **jusqu'à** n **Faire**

(Traitement 1)

FinPour

Structure répétitive avec une condition

TantQue (b - a) > p **Faire**

(Traitement 2)

FinTantQue

Remarque

Le nombre de répétitions dépendra de la condition

- Si la condition n'est pas vérifiée au début alors le "Traitement 2" ne sera pas exécuté du tout.
- Si la condition est vérifiée au début et si la condition n'est pas susceptible d'être modifiée lors du "Traitement 2", alors le "Traitement 2" sera exécuté indéfiniment et l'utilisateur sera obligé d'arrêter lui-même le programme. On dit que le programme boucle indéfiniment, ce qui est une erreur majeure de programmation. Pour que l'algorithme soit correct, il est nécessaire que la condition cesse d'être vérifiée au bout d'un nombre fini de répétitions.

Exemple 1 : Structure répétitive de 1 à n

t **prend la valeur** 0

Pour k **de** 1 **jusqu'à** n **Faire**

x **prend une valeur** aléatoire strictement comprise entre 0 et 1.

u **prend une valeur** aléatoire strictement comprise entre 0 et 1.

y **prend la valeur** (1-x)*u

z **prend la valeur** 1-x-y

Si x <0.5 **et** y<0.5 **et** z<0.5 **alors**

t=t+1

FinSi

FinPour

Exemple 2 : Structure répétitive avec une condition

TantQue (b - a) >p **Faire**

m **prend la valeur** (b + a) / 2

Si f (a)*f(m) <0 **alors**

b **prend la valeur** m

Sinon

a **prend la valeur** m

FinSi

FinTantQue

3.2.2.2 Structures répétitives en langage Matlab

- ***Exemple 1 :*** Structure répétitive de 1 à n

```
t=0;
n=10;
for k=1:n
x= rand () ;
y=(1-x)*rand();
z=1-x-y ;
    if x < 0.5 & y < 0.5 & z< 0.5
    t=t+1
    end

end
```

Exemple 2 : Structure répétitive avec une condition

```
while (b - a) >p
m = (b + a) / 2 ;
    if f (a)*f(m) <0
    b = m ;
    else
    a = m ;
    end
end
```

4 Le logiciel MatLab

4.1 Présentation du logiciel MatLab [3]

MATLAB (Matrix Laboratory) est un environnement de calcul numérique et un langage de programmation développé par MathWorks. Il est largement utilisé dans les domaines de l'ingénierie, des sciences appliquées et de l'analyse de données pour des applications telles que le traitement du signal, la modélisation, la simulation et l'optimisation.

MATLAB est conçu pour manipuler facilement des matrices et des tableaux, ce qui le rend particulièrement puissant pour l'algèbre linéaire. Il propose de nombreuses fonctions intégrées pour les calculs numériques, l'analyse de données, la résolution d'équations différentielles et la visualisation des résultats. Son interface graphique interactive permet la création de scripts et d'interfaces graphiques pour automatiser les tâches et représenter les résultats sous forme de graphiques 2D et 3D. De plus, le logiciel inclut Simulink, une extension permettant de modéliser et simuler des systèmes dynamiques sous forme de blocs. MATLAB peut également interagir avec des langages comme Python, C, C++ et Java pour l'intégration de codes externes.

Dans le domaine du génie civil, MATLAB est utilisé pour la modélisation des structures, la simulation des matériaux et l'analyse des données expérimentales. Il sert aussi au traitement du signal et de l'image, à la reconnaissance de formes et au filtrage numérique. En intelligence artificielle et machine learning, il permet le développement et l'entraînement de modèles d'apprentissage automatique et de réseaux de neurones. Il est également employé pour la dynamique des fluides, notamment pour la résolution d'équations aux dérivées partielles et la

simulation des écoulements. Enfin, MATLAB est un outil puissant pour l'optimisation et la recherche opérationnelle, avec des applications dans l'industrie et la logistique.

Parmi ses avantages, MATLAB se distingue par sa facilité d'utilisation grâce à une interface conviviale et une documentation détaillée. Il offre une grande puissance de calcul et une gestion efficace des matrices et des algorithmes complexes. De plus, il bénéficie d'une large communauté et d'un support technique actif qui assurent une assistance et des mises à jour régulières. Toutefois, son principal inconvénient réside dans son coût élevé, bien que des licences académiques soient disponibles. Il est également moins performant pour le calcul parallèle par rapport à certains outils comme Python avec TensorFlow ou C++.

MATLAB est ainsi un outil incontournable pour les ingénieurs et chercheurs, notamment dans le génie civil, le traitement des matériaux et la modélisation des structures.

4.2 Principaux éléments de MatLab [4]

MATLAB repose sur plusieurs éléments clés qui facilitent le calcul numérique, la modélisation et la programmation.

Le **bureau MATLAB (MATLAB Desktop)** est l'environnement de travail principal. Il comprend plusieurs fenêtres essentielles comme l'**éditeur de script**, la **fenêtre de commande**, l'**espace de travail** qui affiche les variables en mémoire, et l'**historique des commandes** qui permet de consulter les commandes précédemment exécutées.

Le **langage MATLAB** est basé sur la manipulation des matrices et des tableaux. Il permet d'exécuter des opérations mathématiques avancées, d'écrire des scripts (.m) et des fonctions pour automatiser des tâches.

Les **fonctions et bibliothèques intégrées** offrent des outils pour le calcul numérique, la résolution d'équations différentielles, l'optimisation, le traitement du signal, le traitement d'images, et l'apprentissage automatique.

Le **graphique et la visualisation des données** permettent d'afficher des résultats sous forme de courbes 2D, de graphes 3D et de représentations interactives grâce à des commandes comme `plot`, `surf` ou `scatter3`.

Simulink est une extension de MATLAB qui permet de modéliser et simuler des systèmes dynamiques à l'aide de blocs graphiques, très utilisé en ingénierie et en contrôle des systèmes.

Le **gestionnaire d'applications MATLAB App Designer** permet de créer des interfaces graphiques interactives (GUI) sans avoir besoin de programmer en profondeur.

Enfin, MATLAB supporte l'**interopérabilité avec d'autres langages** comme Python, C, C++, et Java, permettant d'intégrer des algorithmes développés dans d'autres environnements.

4.3 Les instructions du langages MatLab [3]

Le langage MATLAB repose sur des instructions et des commandes spécifiques pour exécuter des opérations mathématiques, manipuler des matrices, contrôler le flux d'exécution et visualiser des données.

Les **opérations de base** incluent l'addition (`+`), la soustraction (`-`), la multiplication (`*`), la division (`/` ou `\`), et l'exponentiation (`^`). Les opérations élément par élément sont effectuées avec `.*`, `./`, et `.^`.

Les **commandes de gestion des variables** incluent `clc` pour nettoyer la fenêtre de commande, `clear` pour supprimer les variables en mémoire, et `who` ou `whos` pour afficher les variables existantes.

Les **structures de contrôle** incluent les boucles **`for`** et **`while`** pour l'exécution répétitive, ainsi que les instructions conditionnelles **`if`**, **`elseif`**, et **`else`**.

La manipulation des matrices et tableaux est essentielle en MATLAB. On peut créer une matrice avec `A = [1 2; 3 4]`, accéder à un élément avec `A(1,2)`, extraire une colonne avec `A(:,2)`, et effectuer des opérations matricielles comme l'inversion avec `inv(A)` ou le déterminant avec `det(A)`.

Les **fonctions et scripts** permettent d'automatiser des tâches. Un script est enregistré dans un fichier `.m`, tandis qu'une fonction se définit avec `function [out] = nom_fonction(in)`.

Les **commandes de visualisation** incluent `plot(x, y)` pour tracer une courbe, `bar(x)` pour un diagramme en barres, `hist(x)` pour un histogramme, et `surf(X, Y, Z)` pour des graphes 3D.

Enfin, MATLAB permet d'exécuter des opérations avancées comme la résolution d'équations avec `solve()`, le calcul numérique d'intégrales avec `integral()`, et la transformation de Fourier avec `fft()`…etc

Chapitre 1 : Résolution d'équations non linéaires

Dans ce chapitre, nous présentons quelques méthodes pour trouver les racines d'une équation non linéaire de la forme $f(x) = 0$. Résoudre cette équation, déterminer les zéros de f, ou encore identifier les pôles de $1/f$ sont des formulations équivalentes. De telles problématiques apparaissent fréquemment, que ce soit pour déterminer le point de fonctionnement d'une diode à partir de sa caractéristique, la concentration d'une espèce chimique dans un mélange réactionnel, ou encore la fréquence de coupure d'un filtre électrique. Nous nous limiterons ici à l'étude des fonctions réelles. Selon les besoins, il peut s'agir de chercher toutes les solutions de $f = 0$, quelques-unes seulement, ou une en particulier, comme la plus petite. Un cas particulier notable est celui où f est un polynôme, pour lequel les racines sont soit réelles, soit complexes conjuguées par paires, leur nombre étant égal au degré du polynôme.

Dans la plupart des cas, il est impossible d'exprimer une solution analytique de l'équation $f = 0$. Cela n'est envisageable que pour les polynômes de degré inférieur ou égal à 4 ou pour certaines fonctions simples. Par conséquent, les méthodes générales de recherche de racines sont essentiellement itératives. À partir d'une solution approximative initiale, elles produisent une série d'approximation de plus en plus précises. Il est donc crucial d'étudier la convergence des méthodes employées, leur rapidité, ainsi que de définir un critère d'arrêt des itérations. Il convient également de tenir compte des erreurs d'arrondi inévitables dans tout calcul numérique. Il est important de noter qu'aucune méthode connue ne fonctionne efficacement de manière totalement « aveugle » : il est nécessaire de disposer d'au moins une estimation approximative de l'emplacement de la racine. Ainsi, il est fortement recommandé de tracer le graphe de la fonction pour se faire une idée du nombre et de la position des zéros. Cette tâche est aujourd'hui grandement facilitée par les calculatrices modernes.

Les algorithmes de recherche des racines s'appuient plus ou moins directement sur le théorème suivant.

Théorème des valeurs intermédiaires – Soit f une fonction continue dans $[a, b]$; alors, pour tout réel F compris entre $f(a)$ et $f(b)$, il existe au moins un réel c de $[a, b]$ tel que $f(c) = F$.

En d'autres termes, si $f(a)$ et $f(b)$ sont de signes contraires, la fonction continue f présente au moins un zéro entre a et b.

1 Méthode de bissection ou de dichotomie [5]

Nous nous intéressons à une fonction $f(x)$ continue sur l'intervalle $I = [a, b]$ et telle que $f(a) \times f(b) < 0$; en d'autres termes, nous avons « encadré » la racine. Il découle de ces hypothèses que f s'annule au moins une fois dans I et c'est ce zéro que nous souhaitons localiser précisément. Voici un fragment de programme (en Matlab) correspondant à l'algorithme de bissection.

Nous considérons une fonction $f(x)$ continue sur l'intervalle $I = [a, b]$ et vérifiant $f(a) \times f(b) < 0$, ce qui signifie que la racine est « encadrée » dans cet intervalle. Ces hypothèses garantissent que f s'annule au moins une fois dans I, et notre objectif est de localiser ce zéro avec précision.

Le principe de cette méthode est simple : on calcule la valeur de la fonction au milieu (x_m) de l'intervalle $[x_g, x_d]$ et on observe le changement de signe. Si le signe change entre x_g et x_d, la racine se situe dans cet intervalle, et on redéfinit la borne droite. Si le changement de signe se produit entre x_m et x_d, alors x_m devient la nouvelle borne gauche. À chaque itération, la longueur de l'intervalle contenant la racine est divisée par deux. Cependant, cette ébauche d'algorithme est incomplète : elle ne prévoit pas le cas où $f(x_m) = 0$ exactement. Quelles conséquences peut-on envisager si $a > b$ ou si l'intervalle ***I*** contient plusieurs racines ?

Comme pour toute méthode itérative, un critère d'arrêt est nécessaire. Ici, il s'agit de vérifier que la longueur du segment $[x_g, x_d]$ devient inférieure à une constante « **`ecart`** ». Toutefois, cette condition pourrait ne jamais être satisfaite, ce qui impose de surveiller également le nombre d'itérations. Le calcul est arrêté si « **`iterations`** » dépasse la limite fixée « **`max_iterations`** ». De manière générale, une méthode itérative génère une suite d'approximations successives de la racine $\{x^{(1)}, x^{(2)}, \dots, x^{(N)}, \dots\}$, pour laquelle divers critères de convergence peuvent être envisagés, comme par exemple :

(a) $|x^{n+1} - x^n| < \varepsilon_x$;
(b) (b) $\left|f(x^{(n)})\right| < \varepsilon_y$;
(c) (c) $\left|\frac{x^{(n+1)} - x^{(n)}}{x^{(n)}}\right| < \varepsilon_r$;
(d) (d) $\left|f\left(x^{(n+1)}\right) - f\left(x^{(n)}\right)\right| < \varepsilon_s$

Les options **(a)** et **(b)** sont les plus courantes.

Une fois optimisé, l'algorithme de dichotomie offre plusieurs avantages. Il garantit la convergence tant que les conditions initiales sont respectées, il est très simple à programmer, et le calcul de f n'a pas besoin d'une grande précision, car seule l'information sur le signe de la fonction est utilisée, et non sa valeur exacte. Cependant, l'inconvénient majeur de cette méthode réside dans sa lenteur. L'intervalle contenant la solution est divisé par deux à chaque itération, ce qui signifie que la convergence est globalement linéaire.

2 Méthode du point fixe [6]

Nous cherchons à résoudre l'équation suivante :

$$x = g(x)$$

Toute solution de cette équation est appelée un point fixe de l'application g, c'est-à-dire un point où x est égal à son image par g. Pour y parvenir, nous utilisons un processus d'itérations successives, en partant d'une estimation initiale $x^{(0)}$ de la solution :

$$x^{(n+1)} = g(x^{(n)})$$

Cette équation (5.2) possède une interprétation géométrique intuitive. La solution $x^{(*)}$ de l'équation $x = g(x)$ correspond au point d'intersection entre la première bissectrice (la droite d'équation $y = x$ et la courbe $y = g(x)$.

Lors de chaque itération, l'ordonnée $g(x^{(0)})$ devient la nouvelle abscisse $x^{(1)}$. Le point $\left(x^{(1)}, 0\right)$ est obtenu à partir de $\left(x^{(0)}, g(x^{(0)})\right)$ par une réflexion orthogonale par rapport à la droite $y = x$. Ce processus se répète pour obtenir $x^{(2)}$, $x^{(3)}$, ..., etc.

En appliquant cette méthode à différentes fonctions g, vous observerez deux comportements possibles pour la suite $x^{(0)}$, $x^{(1)}$, ..., : soit elle converge vers un point fixe attractif, soit elle diverge vers un point fixe répulsif, comme illustré par les figures (Figure 1, Figure 2).

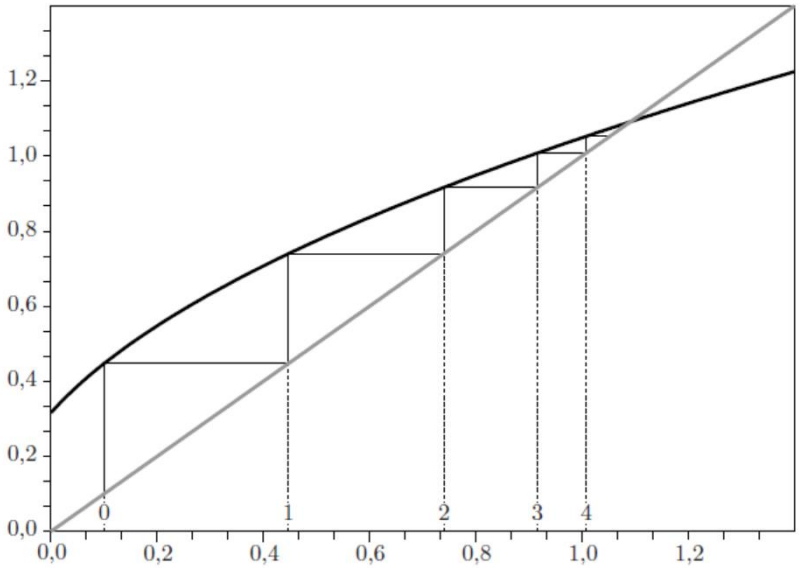

Figure 1. Itération convergente [6]

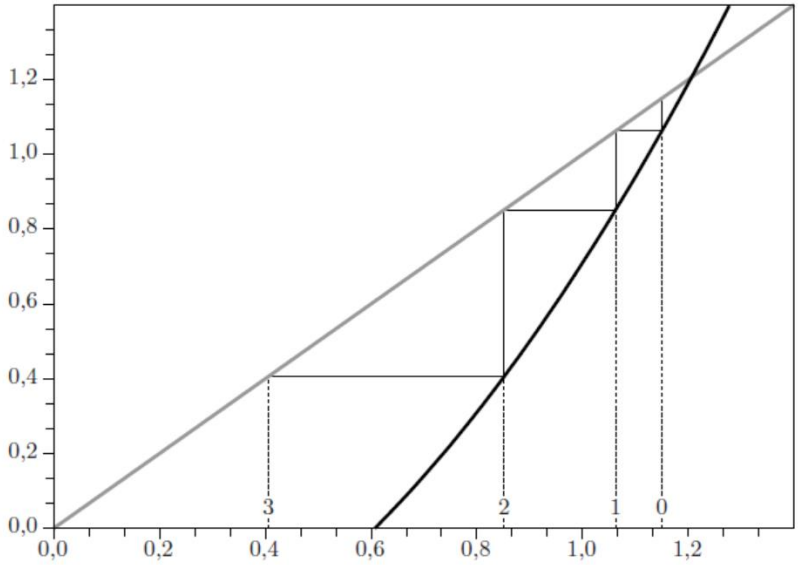

Figure 2. Itération divergente [6]

Cela soulève la question suivante : sous quelle(s) condition(s) la suite $x^{(n)}$ converge-t-elle, et vers quelle valeur ? Le raisonnement, bien que simple dans ce cas, est représentatif de nombreuses analyses de convergence. Supposons que la fonction g et sa dérivée g' soient continues sur un intervalle $I = [a, b]$ contenant la racine x^*. De plus, g doit satisfaire la condition $a \leq g(x) \leq b$ pour tout $x \in I$. Cette hypothèse garantit que tous les termes $x^{(k)}$ restent dans I, à condition que l'approximation initiale $x^{(k)}$ appartienne également à cet intervalle.

La racine x^* vérifie alors l'équation suivante :

$$x^* = g(x^*)$$

Introduisons l'erreur à l'itération k

$$e_k \equiv x^{(k)} - x^*$$

L'erreur à l'itération suivante, $\boldsymbol{k+1}$, peut être exprimée simplement en fonction de e_k

$$e_{k+1} = x^{(k+1)} - x^* = g\left(x^{(k)}\right) - g(x^*)$$

En appliquant le théorème des accroissements finis, le second membre devient :

$$g\left(x^{(k)}\right) - g(x^*) = \left(x^{(k)} - x^*\right) g'(\xi_k)$$

où ξ_k est un point compris entre x^* et $x^{(k)}$. La relation obtenue s'écrit alors :

$$e_{k+1} = g'(\xi_k) e_k$$

Pour que l'itération converge, il est nécessaire que l'erreur tende vers zéro lorsque $\boldsymbol{k \to \infty}$. Cela sera le cas si $|g'(\xi_k)| < 1$ pour tout k. Ainsi, on obtient le théorème suivant.

Théorème : Soit une fonction g continûment dérivable sur l'intervalle $[a, b]$, vérifiant $g(x) \in [a, b]$ pour tout $x \in [a, b]$. Supposons qu'il existe une constante M telle que

$$M = \sup_{a \le x \le b} |g'(\xi_k)| < 1$$

Alors, pour toute valeur initiale $x^{(k)} \in [a, b]$, la suite des itérés $x^{(0)}, x^{(1)}, \dots, x^{(n)}$ converge vers la solution de l'équation $x = g(x)$. De plus, on a :

$$\lim_{n \to \infty} \frac{x^* - x^{(n+1)}}{x^* - x^{(n)}} = g'(x^*)$$

Ainsi, l'erreur est réduite à chaque itération par un facteur strictement inférieur à 1, ce qui caractérise une convergence dite **linéaire**.

La méthode du point fixe est relativement générale, car de nombreuses équations peuvent être réécrites sous la forme $x = g(x)$. Un exemple classique, remontant à l'Antiquité, est l'équation $x^2 = a$, qui peut être reformulée en $x = a/x$ ou $x = \frac{1}{2}(x + a/x)$. Les itérations correspondantes sont toujours utilisées pour obtenir une approximation de $\sqrt{a}$.

La Figure 3 illustre l'application de la méthode du point fixe à l'équation $x = \sqrt{\cos x}$. Les conditions de convergence sont satisfaites sur l'intervalle [0,1], et la méthode converge effectivement vers $x^* = 0.824132$. Afin de préserver la clarté de la figure, le nombre d'itérations a été limité à 5. En pratique, un critère de convergence tel que $\left|x^{(n)} - x^{(n-1)}\right| < \varepsilon_x$ aurait été défini.

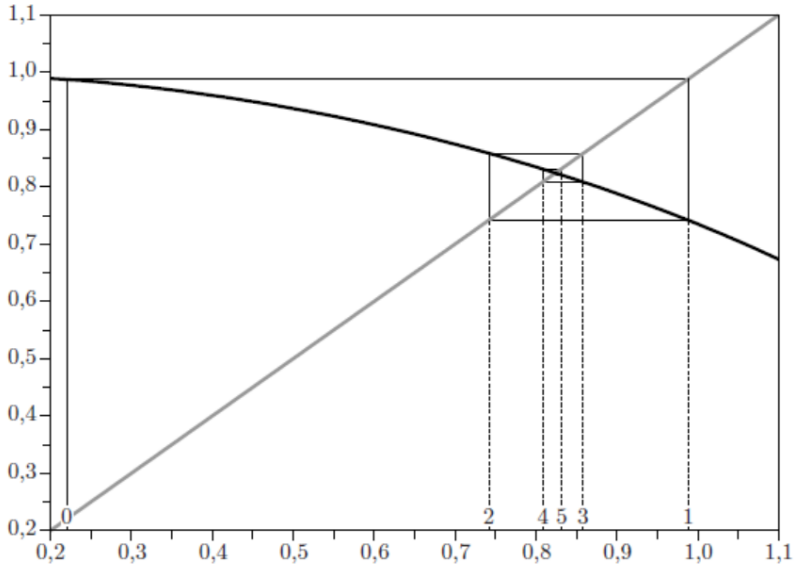

Figure 3 . Résolution de $x = \sqrt{cos\ x}$ par la méthode du point fixe [6].

3 Méthode de Newton [6]

La méthode de Newton est utilisée pour résoudre une équation de la forme $f(x) = 0$. À partir d'une approximation initiale $x^{(0)}$ de la racine, nous traçons la tangente à la courbe $y = f(x)$ au point $x^{(0)}$. Cette tangente coupe l'axe des abscisses en $x^{(1)}$, puis une nouvelle tangente est tracée à $x^{(1)}$, et ainsi de suite, en itérant le processus jusqu'à convergence. L'application de cette méthode à l'équation $x^2 = \cos x$ est illustrée à la **Erreur ! Source du renvoi introuvable.**, où nous avons démarré avec $x^{(0)} = 0.22$ et effectué 5 itérations.

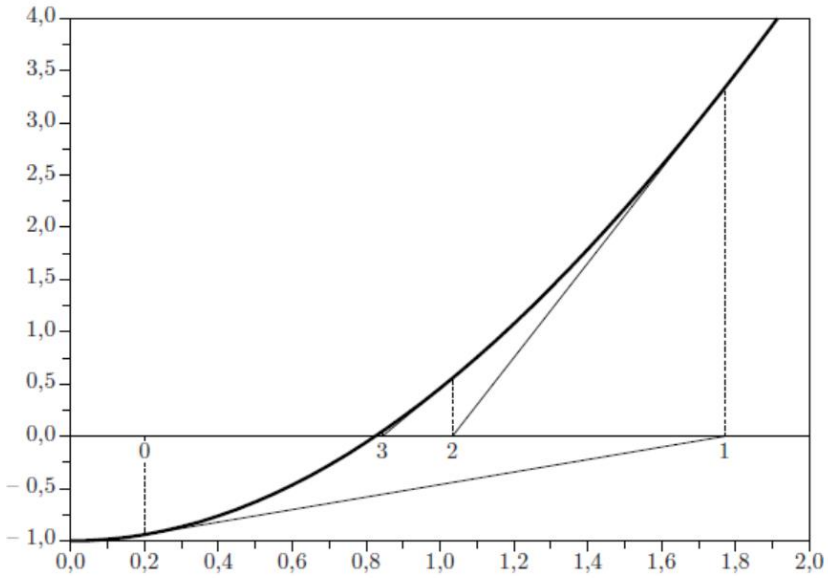

Figure 4. Résolution de $x^2 - \cos x = 0$ par la méthode de Newton [6]

Formulons mathématiquement la description précédente. L'équation de la droite de pente $f'\left(x^{(k)}\right)$ passant par le point $\left(x^{(k)}, f\left(x^{(k)}\right)\right)$ est donnée par : $y - f\left(x^{(k)}\right) = f'\left(x^{(k)}\right)\left(x - x^{(k)}\right)$.

Cette droite coupe l'axe des x en un point d'abscisse :

$$x^{(k+1)} = x^{(k)} - \frac{f\left(x^{(k)}\right)}{f'\left(x^{(k)}\right)}$$

Eq. 1

Cette équation définit l'algorithme de Newton pour résoudre les équations non linéaires. On remarque immédiatement que la méthode diverge lorsque f' est nul ou trop faible. En effet, si la pente de la courbe en x_k est faible, le point d'intersection (d'abscisse $x^{(k+1)}$) de la tangente avec l'axe des x sera très éloigné, ce qui risque de faire diverger l'algorithme.

Pour établir précisément la condition de convergence, observons que si l'on pose :

$$\Phi(x) \equiv x - \frac{f(x)}{f'(x)}$$

l'équation (Eq. 1) devient une itération vers un point fixe :

$$x^{(k+1)} = \Phi\left(x^{(k)}\right)$$

La fonction $\Phi(x)$ est parfois appelée la fonction d'itération de la méthode de Newton. Selon le paragraphe précédent, la convergence dépend de $\Phi'(x)$, et on a :

$$\Phi'(x) = \frac{f(x)f''(x)}{[f'(x)]^2}$$

Soit x^* la racine recherchée, telle que $f(x^*) = 0$, par conséquent

$$\Phi'(x^*) = 0$$

Comme nous supposons que $f, f' \; et \; f''$ sont définies et continues dans la région d'intérêt, il existe un intervalle $I = [x^* - \delta, x^* + \delta]$ tel que :

$$|\Phi'(x)| \leq C < 1$$

Il nous suffit donc de choisir $x^{(0)} \in I$ pour assurer la convergence de l'algorithme de Newton.

Pour estimer la vitesse de convergence, faisons un développement de Taylor de Φ autour de x^* :

$$\Phi\left(x^{(k)}\right) - \Phi(x^*) = \frac{1}{2}\left(x^{(k)} - x^*\right)^2 \Phi''(\eta_k)$$

où η_k est compris entre x^* et $x^{(k)}$. Nous en déduisons que

$$e^{(k+1)} = \frac{1}{2}\left[e^{(k)}\right]^2 \Phi''(\eta_k)$$

Lorsque k croît, $x^{(k)}$ et η_k se rapprochent de x^*, ce que traduit l'expression

$$\lim_{k\to\infty} \frac{e^{(k+1)}}{[e^{(k)}]^2} = \frac{1}{2}\Phi''(x^*)$$

Nous calculons $\Phi''^{(x^*)} = f''^{(x^*)}/f'(x^*)$, d'où le résultat

$$\lim_{k\to\infty} \frac{e^{(k+1)}}{[e^{(k)}]^2} = \frac{f''(x^*)}{2f'(x^*)}$$

Eq. 2

Nous pouvons donc énoncer le théorème suivant.

Théorème – Si $f, f'\ et\ f''$ sont continues dans un voisinage de x^* et si $f(x^*) = 0$ et $f'(x^*) \neq 0$, si $x^{(0)}$ est choisi assez près de x^*, alors la suite définie par l'Eq. 1 converge vers x^*, avec une vitesse de convergence donnée par (Eq. 2) ; la convergence est dite quadratique.

4 TP1 : Résolution d'équations non linéaires

Objectif du TP : Programmation des méthodes de bissection, point fixe et Newton pour le calcul d'une racine approchée d'une fonction « *f* » donnée.

Travail demandé

Soit la fonction suivante : $\boldsymbol{f(x) = 2 * \sin(x) - 1.}$

- Ecrire l'algorithme, pour les trois méthodes, permettant de calculer la racine de la fonction f(x) dans l'intervalle **[-1,1].** Utiliser le critère d'arrêt (nombre d'itérations<50 et tolérance de 0.0001)

- Ecrire un programme MATLAB correspondant à chacune des trois méthodes.

- Dérouler manuellement les programmes.

Solution

Voici un programme (en MATLAB) correspondant à l'algorithme de bissection.

```
clear all, clc
f = @(x) 2*sin(x)-1;            % Définition de la fonction
fplot(f,[-3,3]) % traçage du graphe de la fonction f(x) dans
l'intervalle [-3,3]
grid on
x1=fzero(f,[0,1])                    % Détermination des racines
de cette fonction
x2=fzero(f,[2,3])
Partie2
clear all, clc
f = @(x) 2*sin(x)-1;            % Définition de la fonction à
résoudre
a = -1;                         % Bornes initiales de l'intervalle
b = 1;
tol = 0.00001;                  % Tolérance pour la convergence
ecart =abs(b-a);                     %écat de la solution
max_iterations = 50;            % Nombre maximum d'itérations
if f(a)*f(b)>0                  % Boucle de bissection
error('La fonction doit changer de signe entre a et b.');
else
iterations = 0;                 % Initialisation du nombre
d'itérations
while ecart > tol && iterations < max_iterations
            c = (a + b) / 2;
            ecart=abs(b-a);
           if f(c) == 0         %Évaluation de la fonction au
point milieu. Si f(c) est égal à zéro, alors c est la solution
exacte et le processus est terminé
```

```
disp('la racine est:'), c
            elseif f(c)*f(a)<0
b = c;
            else
a = c;      %Sinon, si f(c) a le même signe que f(a), remplacez a
par c pour le prochain itéré.
            end
            iterations = iterations + 1;
end
end
x_sol = (a + b) / 2; % Calcul de la solution
fprintf('La solution est : %.6f\n', x_sol); % Affichage de la
solution
fprintf('Nombre d''itérations : %d\n', iterations); % Affichage du
nombre d'itérations
```

Voici un programme (en MATLAB) correspondant à l'algorithme du point fixe.

```
clear
f=@(x) 3*x-0.2*exp(x);                          % Définition de la
fonction f(x)
a=0;                                    % Définition de l'intervalle
b=1;
% Définition de la fonction g(x) pour la méthode du point fixe
g = @(x) 0.2*exp(x)/3;
x0 = a;                                 % Valeur initial
tolerance = 1e-4;                       % Tolerance pour la
convergence
maxIterations = 50;                     % Nombre maximal d'itérations
nIterations = 1;
% Boucle d'itérations
if f(a)*f(b)<0
```

```
while abs(x0 - g(x0))  > tolerance && nIterations < maxIterations
    x0 = g(x0); % Application de la fonction g(x)
    nIterations = nIterations + 1;
end
    disp(['La solution approximative est x = ', num2str(x0)]);
    disp(['Nombre d''itérations : ', num2str(nIterations)]);
else
    disp('Pas de racine dans cet intervalle')
end
```

Voici un exemple de programme (en MATLAB) correspondant à l'algorithme de Newton.

```
clear
f=@(x) x^3-2*x-5;
df = @(x) 3*x^2-2;    % Définition de la dérivée de la fonction
f(x)
a=1;   % Intervalle initial [a, b] dans lequel on recherche la
solution
b=3;
x0 = (a+b)/2;
x1 = x0-f(x0)/df(x0);
tolerance = 1e-5;                % Tolerance pour la convergence
maxIterations = 50;              % Nombre maximal d'itérations
nIterations = 0;
% Boucle d'itérations
if f(a)*f(b)<0
    while abs(x0 - x1) > tolerance && nIterations < maxIterations
          x0=x1;
          x1 = x0-f(x0)/df(x0);            % Calcul de x1
          nIterations = nIterations + 1;
    end
    disp(['La solution approximative est x = ', num2str(x0)]);
```

```
    disp(['Nombre d''itérations : ', num2str(nIterations)]);
else
    disp('Pas de racine dans cet intervalle')
end
```

Chapitre 2 : Interpolation et Approximation

1 Approximation de fonctions

L'approximation des fonctions constitue un domaine bien établi des mathématiques, ayant connu un essor significatif dès la fin du XIX^{e} siècle. Dans ce qui suit, nous abordons quelques concepts généraux sur l'approximation, envisagée sous un angle plus mathématique.

Le problème se formule ainsi : nous considérons une fonction $f(x)$ définie sur un intervalle réel, dont nous souhaitons connaître les valeurs en tout point de cet intervalle. Cependant, si f est complexe ou longue à évaluer, il peut être impraticable de calculer toutes les valeurs nécessaires. Une solution consiste alors à remplacer f par une approximation f^*, plus simple à calculer. Pour que cette substitution ait un sens, plusieurs questions se posent : dans quel ensemble de fonctions choisir f^* ? Quel critère permettra de juger la qualité de cette approximation ? Et sur quel intervalle ces critères doivent-ils être respectés ?

Nous nous limiterons ici au cas de l'approximation polynômiale, où f^* est un polynôme de degré au plus n (c'est-à-dire une combinaison linéaire des termes x^k pour $k = 0, 1, 2, \dots, n$). Le théorème de Weierstrass garantit, sous des conditions assez générales, l'existence d'un tel polynôme f^*.

Théorème : Soit f une fonction continue sur l'intervalle fini $I = [a, b]$, et ε un réel strictement positif donné. Il existe alors un polynôme f^* tel que :

$$\sup_{x \in I} |f(x) - f^*(x)| < \varepsilon$$

Il est à noter que le théorème ne précise aucune méthode pour construire f^*. Dans ce contexte, la « distance » entre f et f^* est définie par la « norme infinie » :

$$\|f - f^*\| = \sup_{x \in I} |f(x) - f^*(x)|$$

Tschebychef et de la Vallée-Poussin ont travaillé à élaborer le ***polynôme d'approximation "minimax"***, qui réduit au minimum la distance maximale (selon la norme infinie) entre f et f^*.

2 Polynôme de Tchebychev [7]

Ces polynômes se définissent simplement par :

$$T_n(x) = \cos[n \arccos(x)]$$

Une relation de récurrence reliant $T_{n-1}, T_n \, et \, T_{n+1}$ peut être établie en développant $\cos[(n \pm 1)x]$:

$$T_{n+1}(x) + T_{n-1}(x) = 2xT_n(x)$$

À partir de cette relation, on peut déduire les expressions des premiers polynômes.

$T_0 = 1$; $T_1 = x$; $T_2 = 2x^2 - 1$; $T_3 = 4x^3 - 3x$;

$$T_4 = 8x^4 - 8x^2 + 1$$

Les polynômes T_n satisfont l'équation différentielle suivante :

$$(1 - x^2)T_n'' - xT_n' + n^2T_n = 0$$

Ils sont orthogonaux sur l'intervalle $[-1,1]$ avec la fonction de poids :

$$\omega = \frac{1}{\sqrt{1 - x^2}}$$

Leur fonction génératrice est donnée par :

$$g(u, x) = \frac{1 - ux}{1 - 2ux + u^2} = \sum_{k=0}^{\infty} u^k T_k(x)$$

3 Interpolation [8]

Soit f une fonction donnée, $f^*(x; a_0, a_1, \dots, a_n)$ une fonction dépendant de $n + 1$ paramètres $a_0, a_1, \dots, a_n$ ainsi qu'un ensemble d'abscisses $x_0, x_1, \dots, x_n$, Le problème d'interpolation consiste à déterminer les a_k de manière à satisfaire les $\boldsymbol{n + 1}$ équations suivantes :

$f^*(x_k; a_0, a_1, \dots, a_n) = f(x_k), \qquad k = 0,1,2, \dots, n$

où $f(x_k)$ est noté de manière équivalente f_k. Chaque paire (x_k, f_k) correspond à un point du plan appelé pivot ou nœud.

Il convient ensuite de définir la classe de fonctions à laquelle appartient f^*. Le critère essentiel est la dépendance de f^* vis-à-vis des paramètres a_k. On distingue deux cas principaux : les fonctions où f^* dépend linéairement des a_k et celles où cette dépendance est non linéaire. Une forme très générale pour f^* est donnée par :

$$f^*(x) = \sum_0^n a_k \varphi_k(x)$$

Les propriétés de l'interpolation, telles que la facilité de calcul ou la sensibilité aux erreurs d'arrondi, dépendent du choix des *fonctions de base* $\varphi_k(x)$. Parmi les approches courantes, on trouve l'interpolation polynomiale.

$$f^*(x) = a_0 + a_1 x + a_2 x^2 + \cdots, + a_n x^n$$

et l'interpolation trigonométrique

$$f^*(x) = a_0 + a_1 e^{ix} + a_2 x^{2ix} + \cdots, + a_n x^{inx}$$

relèvent de ce cas. Par contre, l'interpolation rationnelle

$$f^*(x; a_0, a_1, \dots, a_m, b_0, b_1, \dots, b_n) = \frac{a_0 + a_1 x + a_2 x^2 + \cdots, + a_m x^m}{b_0 + b_1 x + \cdots, + b_n x^n}$$

(L'interpolation rationnelle, nécessitant $m + n$ coefficients, dépend non linéairement des b_n. Bien que parfois employée, nous nous concentrerons ici sur l'interpolation polynomiale, qui ne requiert que des calculs simples.

Contrairement à l'approximation, il ne semble pas immédiatement nécessaire de définir un intervalle d'application pour l'interpolation. Toutefois, cette définition est implicite. Si x_{min} désigne le plus petit des x_i et x_{max} le plus grand, nous parlons d'interpolation lorsque

$x_{min} \leq x \leq x_{max}$. En revanche, si x se situe en dehors de cet intervalle $[x_{min}, x_{max}]$, il s'agit d'extrapolation.

4 Interpolation de Newton (Le polynôme de Newton) [6]

4.1 Interpolation linéaire

Revenons à l'interpolation linéaire. Dire que la fonction $f(x)$ est bien représentée par une interpolation linéaire dans un certain intervalle revient à affirmer que le rapport

$$\frac{f(x_1) - f(x_0)}{x_1 - x_0}$$

(la pente moyenne de la courbe représentative) est presque constant, indépendamment de x_0 et x_1, dans cet intervalle. Ce rapport, appelé différence divisée du premier ordre relative à x_0 et x_1, se note généralement

$$f[x_0, x_1] \equiv \frac{f(x_1) - f(x_0)}{x_1 - x_0}$$

Notez que $f[x_0, x_1] = f[x_1, x_0]$. Nous pouvons également écrire l'approximation

$$f[x_0, x] \cong f[x_0, x_1]$$

Eq. 3

Ou encore

$$f(x) \cong f(x_0) + (x - x_0) f[x_0, x_1]$$

Eq. 4

Le second membre de cette dernière équation correspond au polynôme du premier degré qui interpole $\boldsymbol{f}$ sur les pivots (x_0, x_1). Nous noterons cette expression $p_{0,1}$ dans la suite. Il est également utile d'introduire les notations suivantes.

$$p_0(x) \equiv f[x_0] \equiv f(x_0)$$

Ainsi, $f[x_0]$ représente la différence divisée d'ordre zéro, et $p_0(x)$ est le polynôme d'interpolation de degré zéro, coïncidant avec f en x_0. La formule (Eq. 4) peut alors être réécrite sous une forme qui se prêtera facilement à une généralisation ultérieure :

$$p_{0,1} = f[x_0] + (x - x_0)f[x_0, x_1]$$

Eq. 5

À moins que f soit strictement linéaire en $\boldsymbol{x}$, la pente de la sécante $f[x_0, x_1]$ dépendra des abscisses x_0 *et* x_1. Si $\boldsymbol{f}$ est une fonction quadratique en x, la pente de la sécante reliant les points d'abscisses x_0 et x devient elle-même une fonction linéaire de x, lorsque x_0 est fixé. On peut alors s'attendre à ce que l'expression

$$\frac{f[x_1, x_2] - f[x_0, x_1]}{x_2 - x_0}$$

soit indépendante des trois abscisses x_0, x_1 et x_2. En général, ce rapport est appelé **différence divisée d'ordre 2** relative à x_0, x_1 et x_2, et il se note $f[x_0, x_1, x_2]$. Cette expression conserve également une symétrie par rapport à ses arguments. Nous pouvons alors reformuler la relation (Eq. 3).

$$f[x_0, x] - f[x_0, x_1] = f[x_0, x] - f[x_1, x_0] = (x - x_1)f[x_0, x_1, x]$$

d'où nous déduisons la relation exacte qui remplace (Eq. 5)

$$f(x) = f(x_0) + (x - x_0)f[x_0, x_1] + (x - x_0)(x - x_1)f[x_0, x_1, x]$$

Telle quelle, cette formule est peu utile, car connaître $f[x_0, x_1, x]$ revient à connaître $f(x)$, ce que nous cherchons justement à éviter. Mais l'erreur $E(x)$ commise en remplaçant $f(x)$ par $p_{0,1}$ est donnée par

$$f - p_{0,1} = (x - x_0)(x - x_1)f[x_0, x_1, x]$$

Un résultat que nous généraliserons plus tard.

4.2 Les différences divisées

Les différences divisées des ordres $0, 1, 2, \ldots, \boldsymbol{k}$ sont définies de manière récurrente par les relations suivantes :

$$f[x_0] = f(x_0), \qquad f[x_0, x_1] = \frac{f[x_1] - f[x_0]}{x_1 - x_0},$$

$$f[x_0, \ldots, x_k] = \frac{f[x_1, \ldots, x_k] - f[x_0, \ldots, x_{k-1}]}{x_k - x_0}$$

Eq. 6

Dans chaque numérateur, les $k-1$ premiers arguments du premier terme sont identiques aux $k-1$ derniers arguments du second terme, tandis que le dénominateur correspond à la différence entre les arguments distincts des deux termes. On peut démontrer par récurrence que $f[x_0, \ldots, x_k]$ est une fonction entièrement symétrique de ses $k+1$ arguments, ce qui rend leur ordre indifférent. Il en découle que $f[x_0, x_1, \ldots, x_k]$ peut être exprimée comme le quotient de la différence entre deux différences divisées d'ordre $k-1$ (ayant x_0 arguments communs), divisé par la différence entre les arguments distincts :

$$f[x_0, x_1, x_2, x_3] = \frac{f[x_0, x_1, x_2] - f[x_1, x_2, x_3]}{x_0 - x_3} = \frac{f[x_0, x_2, x_3] - f[x_1, x_2, x_3]}{x_0 - x_1}$$

Jusqu'à présent, nous avons supposé que les abscisses étaient distinctes. Cependant, si deux abscisses sont égales, il est possible de donner un sens à la différence divisée en passant à la limite :

$$\lim_{h \to 0} f[x + h, x] = \lim_{h \to 0} \frac{f(x + h) - f(x)}{h} = f'(x)$$

4.3 La formule de Newton

D'après la définition (Eq. 6), nous pouvons écrire, en remplaçant à chaque fois un des x_i par x

$$f(x) = f[x_0] + (x - x_0)f[x_0, x]$$
$$f[x_0, x] = f[x_0, x_1] + (x - x_1)f[x_0, x_1, x]$$
$$\dots = \cdots$$
$$f[x_0, \dots, x_{n-1}, x] = f[x_0, \dots, x_n] + (x - x_n)f[x_0, \dots, x_n, x]$$

En substituant la première équation dans la seconde, nous obtenons l'expression déjà vue

$$\begin{aligned} f(x) = f[x_0] &+ (x - x_0)f[x_0, x_1] + (x - x_0)(x - x_1)f[x_0, x_1, x_2] \\ &+ (x - x_0)(x - x_1) \dots (x - x_{n-1})f[x_0, x_1, \dots, x_n] \\ &+ (x - x_0)(x - x_1) \dots (x - x_n)f[x_0, x_1, \dots, x] \end{aligned}$$

Nous passons de cette expression « formelle » (que nous ne savons pas utiliser puisque nous ignorons le dernier terme) à la formule d'interpolation de Newton en négligeant le terme inconnu :

$$\begin{aligned} p_{0,1,2,\dots,n} = f[x_0] &+ (x - x_0)f[x_0, x_1] + (x - x_0)(x - x_1)f[x_0, x_1, x_2] \\ &+ (x - x_0)(x - x_1) \dots (x - x_{n-1})f[x_0, x_1, \dots, x_n] \end{aligned}$$

L'erreur commise en remplaçant f par p vaut

$$E(x) = (x - x_0)(x - x_1) \dots (x - x_n)f[x_0, x_1, \dots, x]$$

Exemple : Prenons un exemple de calcul de différences divisées. Nous allons interpoler la fonction e^x aux points $[-1; 0{,}5; 1{,}5; 2])$ [6]. Bien que cela ne soit pas indispensable, les abscisses ont été triées par ordre croissant pour faciliter le suivi des étapes. Les pivots sont implicitement numérotés de 0 à 3.

Ordre	0	1	2	3
-1	0,367879			
		1,280842		
		1,5 0,853895		

0,5	1,648721				1,979073				
		2,83297			2,5	0,791629			
		1	2,83297					1,196215	
1,5	4,481689				2,981766			3	0,398738
		2,907367			1,5	1,987844			
		0,5	5,814734						
2	7,389056								

La première colonne présente les valeurs de x, tandis que la deuxième contient celles de f, correspondant aux différences divisées d'ordre zéro. Les colonnes suivantes affichent les valeurs des expressions $x_i - x_j$ et $f_i - f_j$, dont le quotient donne la différence divisée d'ordre 1, reportée dans la colonne numérotée 1. Le calcul se poursuit de manière systématique jusqu'à la différence divisée d'ordre 3, qui est la dernière pouvant être déterminée avec les données disponibles.

Le polynôme de Newton de degré 3 s'exprime ainsi :

$$p_{0,1,2,3} = 0.367879 + (x - 0.5)0{,}853895 + (x - 0{,}5)(x - 1{,}5)0{,}791629 + (x - 0{,}5)(x - 1{,}5)(x - 2)0{,}398738$$

5 TP 2 : Interpolation et approximation

Objectif du TP :

Programmer et comparer les méthodes d'interpolation de Newton et de Tchebychev pour approcher une fonction donnée à partir de points spécifiques.

Travail à réaliser :

Partie 1 : Méthode de Newton

Soit un ensemble de points (x_i, y_i) dans l'intervalle $[-\mathbf{1}, \mathbf{1}]$**.** Ces points peuvent être arbitrairement choisis.

Ecrire un algorithme permettant de calculer différences divisées pour obtenir le polynôme de Newton.

Implémentez sous l'environnement MATLAB le programme correspondant.

Partie 2 : Méthode de Tchebychev

Ecrire un algorithme de la méthode de Tchebychev en utilisant les points de Tchebychev comme nœuds d'interpolation.

Implémentez sous l'environnement MATLAB le programme correspondant.

Solution

Voici un exemple de programme (en MATLAB) correspondant à l'approximation de la fonction e^x dans l'intervalle $[-1,1]$ en utilisant le polynôme de Tchebychev.

```
% Script MATLAB pour l'approximation de exp(x) avec les polynômes
de Tchebychev

% Définir l'intervalle
a = -1;
b = 1;

% Nombre de termes dans le polynôme de Tchebychev
n = 5; % Ajuster pour améliorer la précision

% Points de Tchebychev sur l'intervalle [-1, 1]
chebyshev_points = cos((2 * (0:n) + 1) * pi / (2 * (n + 1)));

% Rééchelle les points de Tchebychev dans l'intervalle [a, b]
x_chebyshev = 0.5 * (a + b) + 0.5 * (b - a) * chebyshev_points;

% Évaluer la fonction exp(x) sur les points de Tchebychev
f_chebyshev = exp(x_chebyshev);

% Calcul des coefficients de Tchebychev (transformée de Tchebychev
discrète)
coefficients = zeros(1, n + 1);
for k = 0:n
    coefficients(k + 1) = (2 / (n + 1)) * sum(f_chebyshev .* cos(k
* acos(chebyshev_points)));
    if k == 0
```

```
        coefficients(k + 1) = coefficients(k + 1) / 2; % Le
coefficient pour T0 est divisé par 2
    end
end

% Approximation avec le polynôme de Tchebychev
x_eval = linspace(a, b, 100)'; % Points d'évaluation
Tn = zeros(length(x_eval), n + 1);
Tn(:, 1) = 1; % T0(x) = 1
Tn(:, 2) = x_eval; % T1(x) = x

% Construire les polynômes de Tchebychev
for k = 2:n
    Tn(:, k + 1) = 2 * x_eval .* Tn(:, k) - Tn(:, k - 1); %
Relation de récurrence
end

% Approximation f*(x)
f_approx = Tn * coefficients';

% Affichage
figure;
plot(x_eval, exp(x_eval), 'r', 'LineWidth', 1.5); % Fonction exacte
hold on;
plot(x_eval, f_approx, 'b--', 'LineWidth', 1.5); % Approximation
legend('e^x', 'Approximation par Tchebychev');
xlabel('x');
ylabel('f(x)');
title('Approximation de e^x avec les polynômes de Tchebychev');
grid on;
```

Voici un exemple de programme (en MATLAB) correspondant à l'interpolation de la fonction e^x avec le polynôme de Newton

```
% Script MATLAB pour l'interpolation de e^x avec le polynôme de
Newton (sans Symbolic Math Toolbox)

% Données d'interpolation
x_points = [-1, 0.5, 1.5, 2]; % Abscisses des points
y_points = exp(x_points);      % Valeurs de e^x aux abscisses

% Calcul des différences divisées
n = length(x_points);
div_diff = zeros(n, n); % Table des différences divisées
div_diff(:, 1) = y_points'; % Première colonne (f[x0], f[x1], ...)

for j = 2:n
    for i = 1:n-j+1
        div_diff(i, j) = (div_diff(i+1, j-1) - div_diff(i, j-1)) /
...
                         (x_points(i+j-1) - x_points(i));
    end
end

% Coefficients du polynôme de Newton
coefficients = div_diff(1, :);

% Évaluation du polynôme sur un ensemble de points
x_vals = linspace(min(x_points)-1, max(x_points)+1, 100);
y_vals = exp(x_vals);

% Évaluation du polynôme de Newton
```

```
interp_vals = zeros(size(x_vals));
for i = 1:length(x_vals)
    term = coefficients(1);
    product = 1;
    for k = 1:n-1
        product = product * (x_vals(i) - x_points(k));
        term = term + coefficients(k+1) * product;
    end
    interp_vals(i) = term;
end

% Affichage des résultats
figure;
plot(x_vals, y_vals, 'b-', 'LineWidth', 1.5); % Courbe exacte
hold on;
plot(x_vals, interp_vals, 'r--', 'LineWidth', 1.5); % Courbe
interpolée
plot(x_points, y_points, 'ko', 'MarkerSize', 8, 'MarkerFaceColor',
'k'); % Points d'interpolation
legend('e^x (exact)', 'Polynôme de Newton', 'Points
d''interpolation');
xlabel('x');
ylabel('f(x)');
title('Interpolation de e^x avec le polynôme de Newton');
grid on;
```

La Figure 5 illustre le résultat ; comme le polynôme d'interpolation est unique, le résultat final est indiscernable de celui du paragraphe précédent.

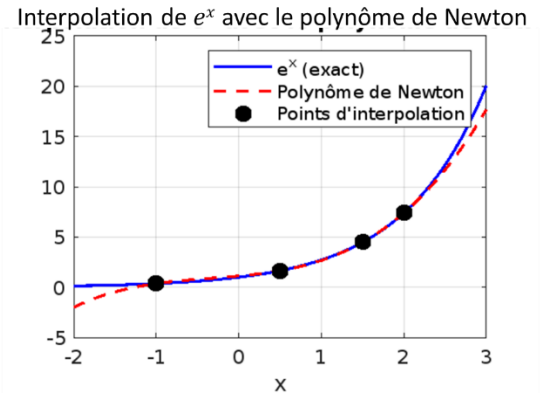

Figure 5. Interpolation par la méthode de Newton

Chapitre 3 : Intégration numérique

Nous nous intéressons désormais au calcul numérique des intégrales, également appelé quadrature numérique [6]. Il est important de distinguer deux cas : celui des intégrales définies, où nous cherchons à déterminer la valeur de l'intégrale :

$$I = \int_a^b f(x)dx$$

en connaissant l'intervalle $[a, b]$ et la fonction f, et celui des intégrales indéfinies (ou primitives), qui consiste à calculer la fonction

$$J(x) = \int_a^x f(u)du$$

dans laquelle nous obtenons une série de valeurs correspondant à la fonction $J(x)$. Ce dernier cas est généralement abordé comme un problème différentiel :

$$J' = f(x)\,; \qquad J(a) = 0$$

Nous allons donc nous concentrer sur l'approximation numérique des intégrales définies. Toutes les méthodes que nous examinerons peuvent être exprimées sous la forme :

$$I = \int_a^b f(x)dx = \sum_1^n \omega_j f(x_j) + E$$

où E représente l'erreur de troncation, les x_j sont les abscisses (ou noeuds), et les ω_j sont les poids. Nous distinguons deux types d'algorithmes. Dans la première classe (méthodes de Newton–Cotes), les abscisses sont équidistantes et les poids sont les seuls paramètres ajustables ; il y en a n, ce qui nous permet de satisfaire n contraintes et de rendre la méthode exacte pour un polynôme de degré $n - 1$. Dans la deuxième classe (méthodes de Gauss), les abscisses et les poids sont tous deux ajustables. Le choix de ces $2n$ paramètres permet de rendre la méthode exacte pour un polynôme de degré $2n - 1$. Bien entendu, les méthodes de Gauss sont plus performantes que celles de Newton–Cotes, car elles offrent une plus grande précision pour un

même temps de calcul, bien qu'elles impliquent une théorie plus complexe et une programmation plus lourde.

1 Méthodes élémentaires d'intégration (Rectangle et Trapèze) [9]

La Figure 6 présente quatre méthodes d'intégration simples.

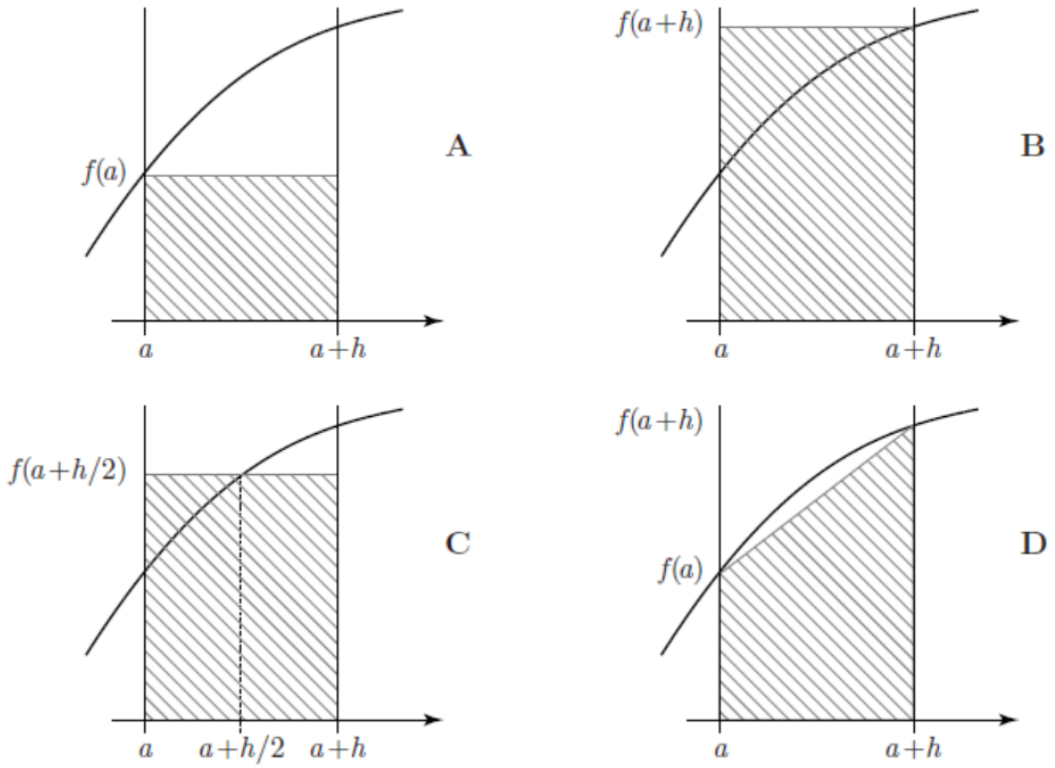

Figure 6. Principe de quatre méthodes simples de quadrature numérique. A : point gauche ou rectangle à droite, B : point droit ou rectangle à gauche, C : « point milieu », D : trapèzes.

(A) et (B) correspondent respectivement à ce que l'on désigne comme la méthode du point gauche et celle du point droit. L'intégrale, représentant l'aire algébrique délimitée par le graphe de f, l'axe Ox, et les verticales d'abscisses $\boldsymbol{a}$ et $\boldsymbol{a+h}$, est approximée par l'aire d'un rectangle ayant la même base et une hauteur égale à $f(a)$ (pour le point gauche) ou à $f(a+h)$ (pour le point droit).

$$\int_a^{a+h} f(x)dx \cong hf(a) \cong hf(a+h)$$

Le théorème de Taylor va nous permettre de borner l'erreur, par exemple dans le cas du rectangle à droite. Il s'écrit

$$f(x) = f(a) + (x-a)f'(\xi)$$

où ξ est un point (fonction de x) de l'intervalle $[a, x]$. Intégrons cette expression entre a et $a+h$.

$$\int_a^{a+h} f(x)dx = \int_a^{a+h} f(a)dx + \int_a^{a+h} f(x-a)f'(\xi)dx$$

Dans l'intervalle d'intégration, $x-a$ est toujours positif ; en supposant que f' est continue dans ce même intervalle, nous pouvons appliquer la formule de la moyenne

$$\int_a^{a+h} f(x)dx = hf(a) + f'(\eta)\int_a^{a+h} f(x-a)dx$$

Où $a \leq \eta \leq a+h$. L'intégrale se calcule à vue après avoir fait le changement de variable $u = x-a$; nous trouvons

$$\int_a^{a+h} f(x)dx = hf(a) + \frac{h^2}{2}f'(\eta)$$

Dans le cas de la figure, $f' > 0$ et la valeur exacte de l'intégrale est en effet plus grande que sa valeur approchée. Appelant M_1 une borne supérieure de $|f'|$ dans l'intervalle $[a, a+h]$, nous pouvons encore écrire

$$\left|\int_a^{a+h} f(x)dx - hf(a)\right| \leq \frac{h^2}{2}M_1$$

Vous pourrez établir une formule analogue dans le cas du point droit. Ces deux méthodes ne sont pas utilisées en pratique, car l'erreur est bien plus grande que celle associée à l'algorithme suivant qui, pourtant, implique un même nombre de calculs de f.

La Figure 6(C) illustre ce que nous appellerons la méthode du « point milieu », par analogie avec le terme anglais. Nous espérons que l'erreur sera plus petite, car elle est la somme des

aires de deux triangles curvilignes qui sont de signes opposés et se compensent approximativement. La formule correspondante s'écrit

$$I \cong hf\left(a+\frac{h}{2}\right)$$

Cherchons l'expression de l'erreur, en supposant $f(x)$ deux fois continûment dérivable dans l'intervalle d'intégration. Nous formons le développement de Taylor de f autour de l'abscisse $a + h/2$

$$f(x) = f\left(a+\frac{h}{2}\right) + \left(x-a-\frac{h}{2}\right) + f'\left(a+\frac{h}{2}\right) + \frac{1}{2}\left(x-a-\frac{h}{2}\right)^2 f''(\xi)$$

Nous intégrons terme à terme entre a et $a + h$

$$I = hf\left(a+\frac{h}{2}\right) + f'\left(a+\frac{h}{2}\right)\int_a^{a+h}\left(x-a-\frac{h}{2}\right)dx + \frac{1}{2}f''(\eta)\int_a^{a+h}\left(x-a-\frac{h}{2}\right)^2 dx$$

Les deux intégrales se calculent aisément en posant $u = x - a - \frac{h}{2}$; la première est nulle, la seconde vaut $h^3/12$. Nous obtenons finalement

$$\int_a^{a+h} f(x)dx = hf\left(a+\frac{h}{2}\right) + f''(\eta)\frac{h^3}{24}$$

Introduisant M_2, une borne supérieure de $|f''|$ sur $[a, a + h]$, nous écrivons aussi bien

$$\left|\int_a^{a+h} f(x)dx - hf\left(a+\frac{h}{2}\right)\right| \leq \frac{h^3}{24}M_2$$

Vous voyez que la compensation entre triangles curvilignes d'aires algébriques opposées est d'autant plus mauvaise que la courbure du graphe (la valeur de f'') est plus grande. L'erreur est ici en h^3, grâce à la symétrie qui a fait disparaître le terme en h^2.

La Figure 6(D) représente un algorithme à peine plus compliqué. La surface cherchée est approchée par l'aire du **trapèze** inscrit, soit

$$\int_a^{a+h} f(x)dx \cong \frac{1}{2}(b-a)[f(a)+f(b)]$$

Une expression dont le terme d'erreur est comparable au précédent.

2 Méthode de Simpson [10]

La **méthode de Simpson** est une méthode d'intégration numérique qui utilise des polynômes quadratiques pour approximer l'intégrale d'une fonction $f(x)$ sur un intervalle donné $[a, b]$. Elle est généralement plus précise que la méthode des trapèzes pour une même subdivision.

2.1 Principe de la méthode de Simpson

La méthode repose sur l'approximation de $f(x)$ par des paraboles dans chaque sous-intervalle.

1. **Subdivision de l'intervalle** : L'intervalle $[a, b]$ est subdivisé en n sous-intervalles égaux, avec n **pair**, de largeur h, où :

$$h = \frac{b-a}{n}$$

2. **Formule de Simpson** : L'intégrale est approximée par :

$$\int_a^b f(x)dx \cong \frac{h}{3}\left[f(a) + 4\sum_{i=1, i\ impair}^{n-1} f(x_i) + 2\sum_{i=2, i\ pair}^{n-2} f(x_i) + f(b)\right]$$

- Les termes aux extrémités ($f(a)$ et $f(b)$) sont pondérés par 1.
- Les termes aux indices impairs sont pondérés par 4.
- Les termes aux indices pairs sont pondérés par 2.

3 TP3 : Intégration numérique

Objectif du TP : Mettre en œuvre et comparer les méthodes du rectangle, du trapèze et de Simpson pour évaluer numériquement une intégrale définie, en programmant ces méthodes en MATLAB

Travail à réaliser

Ecrire un algorithme pour chaque méthode (méthode de rectangle, méthode de trapèze et de Simpson)

Ecrire les programmes MATLAB correspondants.

Solution

Programme Matlab de la méthode des rectangles

```
% Méthode des rectangles : point gauche, milieu et droit

% Définition de la fonction à intégrer
f = @(x) x.^2; % Exemple : f(x) = x^2

% Bornes de l'intégration
a = 0; % Borne inférieure
b = 1; % Borne supérieure

% Nombre de sous-intervalles
n = 10; % Ajustez pour plus de précision

% Largeur des sous-intervalles
h = (b - a) / n;

% Points de discrétisation
x = linspace(a, b, n+1); % Nœuds entre a et b

% Méthode du point gauche
x_gauche = x(1:end-1); % Points gauche des sous-intervalles
I_gauche = h * sum(f(x_gauche));

% Méthode du point droit
x_droit = x(2:end); % Points droit des sous-intervalles
I_droit = h * sum(f(x_droit));

% Méthode du point milieu
```

```
x_milieu = (x(1:end-1) + x(2:end)) / 2; % Points milieu des sous-
intervalles
I_milieu = h * sum(f(x_milieu));

% Affichage des résultats
fprintf('Intégration par la méthode des rectangles :\n');
fprintf('Point gauche : %.6f\n', I_gauche);
fprintf('Point milieu : %.6f\n', I_milieu);
fprintf('Point droit  : %.6f\n', I_droit);

% Affichage graphique
figure;
hold on;
fplot(f, [a, b], 'k', 'LineWidth', 1.5); % Tracé de la fonction
title('Méthodes des rectangles pour l’intégration numérique');
xlabel('x');
ylabel('f(x)');
grid on;

% Point gauche
for i = 1:n
    rectangle('Position', [x_gauche(i), 0, h, f(x_gauche(i))], ...
              'EdgeColor', 'r', 'FaceColor', 'none');
end

% Point droit
for i = 1:n
    rectangle('Position', [x(i), 0, h, f(x_droit(i))], ...
              'EdgeColor', 'g', 'FaceColor', 'none', 'LineStyle',
'--');
end
```

```
% Point milieu
for i = 1:n
    rectangle('Position', [x_gauche(i), 0, h, f(x_milieu(i))], ...
              'EdgeColor', 'b', 'FaceColor', 'none', 'LineStyle',
':');
end

legend('f(x)', 'Point gauche', 'Point droit', 'Point milieu');
hold off;
```

Programme maatlab de la méthode des trapèzes

```
% Script MATLAB : Méthode des Trapèzes pour le calcul de
l'intégrale

% Définir la fonction à intégrer (modifiez ici selon vos besoins)
f = @(x) x.^2; % Exemple : f(x) = x^2

% Définir les bornes d'intégration
a = input('Entrez la borne inférieure a : ');
b = input('Entrez la borne supérieure b : ');

% Nombre de sous-intervalles
n = input('Entrez le nombre de sous-intervalles n : ');

% Calcul du pas
h = (b - a) / n;

% Initialisation de l'intégrale
integrale = 0;
```

```
% Boucle pour appliquer la formule des trapèzes sur chaque sous-
intervalle
for i = 1:n
    % Début et fin du sous-intervalle
    x_i = a + (i-1)*h;
    x_ip1 = x_i + h;

    % Appliquer la formule des trapèzes sur [x_i, x_ip1]
    integrale = integrale + (1/2) * (x_ip1 - x_i) * (f(x_i) +
f(x_ip1));
end

% Afficher le résultat
fprintf('La valeur approximative de l''intégrale est : %.6f\n',
integrale);
```

Voici un script MATLAB pour implémenter la méthode de Simpson :

```
% Définition de la fonction
f = @(x) x.^2; % Exemple : f(x) = x^2

% Définition des bornes de l'intervalle et du nombre de
subdivisions
a = 0; % Borne inférieure
b = 1; % Borne supérieure
n = 10; % Nombre de subdivisions (doit être pair)

% Vérification que n est pair
if mod(n, 2) ~= 0
    error('Le nombre de subdivisions n doit être pair.');
end
```

```
% Calcul de la largeur des sous-intervalles
h = (b - a) / n;

% Points de discrétisation
x = linspace(a, b, n+1);

% Calcul de l'intégrale avec la méthode de Simpson
integrale_simpson = (h / 3) * (f(a) + 4 * sum(f(x(2:2:end-1))) + 2
* sum(f(x(3:2:end-2))) + f(b));

% Affichage des résultats
fprintf('Valeur approchée de l''intégrale par la méthode de Simpson
: %.6f\n', integrale_simpson);
```

Chapitre 4 : Equations Différentielles

De nombreuses questions scientifiques ou techniques s'appuient sur des modèles mathématiques impliquant des équations différentielles. Qu'il s'agisse de la dynamique des corps matériels ou de la cinétique des réactions chimiques, les cours de mathématiques offrent une variété de méthodes analytiques pour résoudre ces équations. On pourrait alors penser que l'analyse numérique n'a pas sa place dans ce domaine. Cependant, la réalité est différente : seule une infime proportion des équations différentielles, en particulier les équations linéaires, peut être résolue analytiquement. Plus les problèmes étudiés s'éloignent des exemples académiques pour se rapprocher de situations réelles, plus les équations deviennent non linéaires et difficiles à résoudre.

Ce chapitre présente plusieurs familles d'algorithmes destinés à résoudre numériquement des problèmes différentiels. Il est essentiel de distinguer une équation différentielle d'un problème différentiel. Une équation différentielle admet une solution générale dépendant de paramètres inconnus (par exemple, n constantes arbitraires pour une équation d'ordre n), rendant impossible une solution numérique directe. En revanche, un problème différentiel associe une équation différentielle à des conditions initiales ou aux limites, ce qui garantit généralement une solution unique. L'objectif de ce chapitre est de montrer comment cette solution peut être approchée numériquement.

On distingue principalement deux types de problèmes différentiels, où l'inconnue est une fonction y. Il y a d'abord les problèmes à condition(s) initiale(s), souvent appelés problèmes de Cauchy, qui peuvent être illustrés par l'exemple suivant :

$$y' = f[x, y(x)]; \qquad y(a) = A; \qquad x \geq a$$

Eq. 7

et les problèmes aux limites, comme par exemple :

$$y'' = g[x, y(x), y'(x)]\ ;\ \ y(a) = A\ ;\ \ \ y(b) = B\ ;\ \ \ a \leq x \leq b$$

Dans ce chapitre, nous nous concentrons exclusivement sur la première catégorie de problèmes : les problèmes à condition(s) initiale(s). Nous supposerons que l'équation

différentielle étudiée est exprimée en fonction de la dérivée d'ordre le plus élevé, comme dans les exemples précédents.

1 Méthode d'Euler [11]

La méthode d'Euler est rarement utilisée en pratique (excepté parfois comme composant dans des méthodes pour résoudre des équations aux dérivées partielles). Cependant, elle constitue une excellente introduction, facilement généralisable dans plusieurs directions intéressantes .

L'algorithme proposé par Euler pour résoudre l'équation (Eq. 7) s'écrit :

$$y_{n+1} = y_n + hf(x_n, y_n) = y_n + hf_n$$

Cette formule peut être interprétée de trois façons :

1. Approximation de la dérivée : La dérivée de y au point x_n peut-être approximée par $y'(x_n) \cong (y_{n+1} - y_n)/h$. En supposant qu'elle est égale à f_n, on obtient la méthode d'Euler.
2. Approximation d'une intégrale : En intégrant les deux membres de l'équation (Eq. 7), on a :

$$y(x_{n+1}) - y(x_n) = \int_{x_n}^{x_{n+1}} f[t, y(t)]dt$$

L'intégrale du second membre est alors approximée par l'aire d'un rectangle de base h et de hauteur f_n (méthode du « point gauche » ou du « rectangle à droite », voir chapitre 3), ce qui donne :

$$y_{n+1} - y_n = hf_n$$

3. Développement de Taylor : Si y_n est connu, on peut approximer y_{n+1} à l'aide d'un développement de Taylor.

$$y(x_n + h) = y(x_n) + hy'_n + \mathcal{O}(h^2)$$

En négligeant le terme d'erreur et en identifiant $y(x_n + h)$, nous retrouvons la formule d'Euler.

2 Méthodes de Runge-Kutta [6]

Les méthodes décrites dans ce paragraphe peuvent être vues comme des extensions de la méthode d'Euler, dans lesquelles la fonction $\boldsymbol{f}$ (le second membre) est évaluée plusieurs fois par pas, afin de diminuer l'erreur de troncature. Ces méthodes figurent probablement parmi les plus couramment utilisées en pratique. Nous présenterons ici la théorie pour une méthode d'ordre 2, bien que, dans les applications, on utilise fréquemment des méthodes d'ordre 4 ou supérieur.

2.1 Méthodes d'ordre 2

Nous cherchons à résoudre le problème défini par l'équation (Eq. 7), en supposant que la solution numérique approchée y_n est connue au point $x_n = x_0 + nh$. Pour cela, nous introduisons une *fonction incrément* $\Phi(x_n, y_n, h)$, telle que

$$y_{n+1} - y_n = h\Phi(x_n, y_n, h), \qquad 0 \leq n \leq N$$

Cette fonction dépend également du second membre f. Considérons par ailleurs la solution exacte $z(t)$ du problème différentiel :

$$z'(t) = f[t, z(t)]; \qquad z(x) = y$$

En termes simplifiés, $\boldsymbol{z(t)}$ représente la solution de l'équation différentielle « passant par le point de coordonnées $(\boldsymbol{x}, \boldsymbol{y})$», où $\boldsymbol{x}$ et y sont considérés comme des constantes arbitraires. L'incrément exact est défini par :

$$z(x + h) - z(x) = h\Delta(x, y, h)$$

Dans le cas du schéma d'Euler, on avait $\boldsymbol{\Phi} = \boldsymbol{f}$. Pour construire une méthode d'ordre 2, nous cherchons une expression de $\boldsymbol{\Phi}$ qui coïncide avec $\boldsymbol{\Delta}$ jusqu'aux termes en h inclus. Un schéma est dit d'ordre p si la différence $\boldsymbol{\Delta} - \boldsymbol{\Phi}$ est de l'ordre $\sigma(h^p)$. En suivant l'approche de Runge et Kutta, nous supposons que $\boldsymbol{\Phi}$ prend la forme :

$$\Phi(x,y,h) = b_1 f(x,y) + b_2 f[x + p_1 h, y + p_2 h f(x,y)]$$

où b_1, b_2, p_1 sont des constantes à déterminer. Pour cela, nous imposons que le développement de Taylor de $\boldsymbol{\Phi}$, au premier ordre en h, coïncide avec celui de $\boldsymbol{\Delta}$ au même ordre. Les développements s'écrivent alors

$$\Delta = f(x,y) + \frac{1}{2}h\left(f_x + f f_y\right) + o(h^2)$$
$$\Phi = (b_1 + b_2)f + h b_2 p_1 f_x + h b_2 p_2 f f_y + o(h^2)$$

D'où trois conditions à remplir pour que $\Delta \equiv \Phi$:

$$\begin{cases} b_1 + b_2 = 1 \\ b_2 p_1 = 1/2 \\ b_2 p_2 = 1/2 \end{cases}$$

Eq. 8

Nous avons quatre paramètres inconnus. Il est courant de conserver un paramètre libre et de déterminer les trois autres à l'aide des relations précédentes. Le paramètre libre est ensuite ajusté pour conférer à l'algorithme une propriété souhaitée, telle qu'une meilleure stabilité ou une réduction des erreurs d'arrondi. Nous décidons ici de garder b_2, que nous renommons β pour la suite. L'équation (Eq. 8) devient alors :

$$\begin{cases} b_1 = 1 - \beta, \\ b_2 = \beta, \\ p_1 = p_1 = \dfrac{1}{2\beta} \end{cases}$$

On obtient ainsi une méthode de Runge-Kutta d'ordre 2, définie par la fonction incrément suivante :

$$\Phi(x,y,h) = (1-\beta)f(x,y) + \beta f\left[x + \frac{h}{2\beta}, y + \frac{h}{2\beta} f(x,y)\right]$$

Parmi toutes les valeurs possibles de β, deux cas se distinguent particulièrement :

- $\beta = \frac{1}{2}$, correspondant à la méthode de Heun (ou Euler améliorée).

- $\beta = 1$, correspondant à la méthode d'Euler modifiée.

Ces algorithmes se formalisent facilement dans leurs expressions respectives

$\beta = \frac{1}{2}$; $\quad y_{n+1} = y_n + \frac{h}{2}\{f(x_n, y_n) + f[x_n + h, y_n + hf(x_n, y_n)]\}$

Et

$\beta = 1$; $\quad y_{n+1} = y_n + hf\left[x_n + \frac{h}{2}, y_n + \frac{h}{2}f(x_n, y_n)\right]$

2.2 Méthodes d'ordre 4

Nous ne détaillerons pas ici les formules des méthodes de Runge-Kutta d'ordre supérieur à 2, car la complexité des calculs augmente rapidement avec l'ordre. Nous nous limiterons à quelques observations générales.

Un schéma de Runge–Kutta à s « étages » est défini par les formules

$$\begin{gathered} k_1 = f(x_n, y_n), \\ k_2 = f(x_n + c_2 h, y_n + h a_{21} k_1) \\ k_3 = f(x_n + c_3 h, y_n + h(a_{31} k_1 + a_{32} k_2)) \\ \cdots \\ k_s = f\left(x_n + c_s h, y_n + h\left(a_{s1} k_1 + \cdots + a_{s,s-1} k_{s-1}\right)\right) \\ y_{n+1} = y_n + h(b_1 k_1 + \cdots + b_s k_s) \\ x_{n+1} = x_n + h \end{gathered}$$

Les valeurs des coefficients k_i dépendent de x et sont donc différentes pour chaque valeur de n. Pour des méthodes explicites, on a $c_1 = 0$ et $a_{sj} = 0$ pour $j \geq s$. On résume souvent cette méthode par le tableau

c_1	0	0	...	0	0
c_2	a_{21}	0	...	0	0
c_3	a_{31}	a_{32}	0	0	0
⋮	⋮	⋮	⋱	⋮	⋮
c_s	a_{s1}	a_{s2}	...	$a_{s,s-1}$	0

	$b_1 \quad b_2 \quad \dots \quad b_{s-1} \quad b_s$

Avec ces notations, la méthode d'Euler modifiée est symbolisée comme

0	$0 \quad 0$
$1/2$	$1/2 \quad 0$
	$0 \quad 1$

Quel est le tableau associé à l'algorithme d'Euler amélioré ? Que se passerait-il si la matrice des a_{sj} n'était pas strictement triangulaire inférieure ?

La méthode de Runge–Kutta « classique » comporte quatre étages et on démontre qu'elle est également d'ordre 4. Elle est définie par le tableau

0	$0 \quad 0 \quad 0 \quad 0$
$1/2$	$1/2 \quad 0 \quad 0 \quad 0$
$1/2$	$0 \quad 1/2 \quad 0 \quad 0$
1	$0 \quad 0 \quad 1 \quad 0$
	$1/6 \quad 2/6 \quad 2/6 \quad 1/6$

Ou par les formules

$$
\begin{aligned}
k_1 &= f(x_n, y_n) \\
k_2 &= f\left(x_n + \frac{h}{2}, y_n + \frac{h}{2}k_1\right) \\
k_3 &= f\left(x_n + \frac{h}{2}, y_n + \frac{h}{2}k_2\right) \\
k_4 &= f(x_n + h, y_n + hk_3) \\
y_{n+1} &= y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \\
x_{n+1} &= x_n + h
\end{aligned}
$$

En pratique, on traite le plus souvent un système différentiel. La « fonction » inconnue y est alors un vecteur, de même que la « fonction » du second membre f. L'expérience montre que la mise en œuvre de la méthode « RK-4 » est alors assez déroutante au début ; il vaut mieux expliciter en détail toutes les formules, quitte à les simplifier par la suite. C'est ce que nous avons fait dans le programme qui suit.

3 TP4 : Equations différentielles

Objectif du TP : il s'agit de développer un algorithme, puis un programme, de résolution numérique de l'équation différentielle suivante :

$$\begin{cases} u'(t) = u(t) \\ u(0) = 1 \end{cases}; \qquad t \in [0;5]$$

Travail demandé :

1. Ecrire les algorithmes permettant de résoudre numériquement l'équation différentielle proposée par les méthodes d'Euler et de Runge-Kutta.
2. Ecrire les programmes MATLAB correspondants à chaque méthode.
3. Représenter sur un même graphique le nuage de points de chaque approximation.

Solution

Voici un exemple de script MATLAB de la méthode d'Euler

```
% Script MATLAB pour résoudre une équation différentielle avec la
méthode d'Euler

% Définir la fonction différentielle dy/dx = f(x, y) sous forme de
fonction anonyme
f = @(x, y) -2 * x * y; % Exemple : dy/dx = -2xy

% Paramètres initiaux
x0 = 0;             % Point initial pour x
y0 = 1;             % Condition initiale y(x0) = y0
h = 0.1;            % Pas d'intégration
xn = 1;             % Point final pour x

% Initialisation
N = (xn - x0) / h; % Nombre d'itérations
x = x0:h:xn;        % Points x discrétisés
y = zeros(1, length(x)); % Vecteur pour stocker les valeurs de y
y(1) = y0;          % Appliquer la condition initiale

% Boucle pour la méthode d'Euler
for n = 1:N
    y(n+1) = y(n) + h * f(x(n), y(n));
end

% Afficher les résultats
fprintf('Valeurs calculées avec la méthode d''Euler :\n');
for i = 1:length(x)
    fprintf('x = %.2f, y = %.4f\n', x(i), y(i));
```

```
end

% Tracer les résultats
figure;
plot(x, y, '-o', 'LineWidth', 1.5, 'MarkerSize', 6);
hold on;

% Tracer la solution exacte si elle est connue (par exemple y =
exp(-x^2) pour dy/dx = -2xy)
y_exact = exp(-x.^2);
plot(x, y_exact, '--', 'LineWidth', 1.5);

xlabel('x');
ylabel('y');
legend('Méthode d''Euler', 'Solution exacte');
title('Résolution de l''équation différentielle par la méthode
d''Euler');
grid on;
```

Exemple de script Matlab des méthodes de Runge-Kutta

```
% Script MATLAB pour implémenter les méthodes de Runge-Kutta
d'ordre 2 et 4

% Définir la fonction à résoudre (f(x, y))
f = @(x, y) -2 * x * y; % Exemple : équation différentielle dy/dx =
-2xy

% Paramètres initiaux
x0 = 0;       % Valeur initiale de x
y0 = 1;       % Valeur initiale de y
h = 0.1;      % Pas de discrétisation
```

```
x_end = 2;    % Valeur finale de x

% Générer les points x
t = x0:h:x_end;
n = length(t);

% Initialisation des résultats pour les deux méthodes
y_rk2 = zeros(1, n);
y_rk4 = zeros(1, n);

% Conditions initiales
y_rk2(1) = y0;
y_rk4(1) = y0;

%% Méthode de Runge-Kutta d'ordre 2 (Heun ou Euler améliorée)
for i = 1:n-1
    k1 = f(t(i), y_rk2(i));
    k2 = f(t(i) + h, y_rk2(i) + h * k1);
    y_rk2(i+1) = y_rk2(i) + (h / 2) * (k1 + k2);
end

%% Méthode de Runge-Kutta d'ordre 4
for i = 1:n-1
    k1 = f(t(i), y_rk4(i));
    k2 = f(t(i) + h / 2, y_rk4(i) + (h / 2) * k1);
    k3 = f(t(i) + h / 2, y_rk4(i) + (h / 2) * k2);
    k4 = f(t(i) + h, y_rk4(i) + h * k3);
    y_rk4(i+1) = y_rk4(i) + (h / 6) * (k1 + 2 * k2 + 2 * k3 + k4);
end

%% Affichage des résultats
```

```
figure;
plot(t, y_rk2, 'r-o', 'DisplayName', 'Runge-Kutta Ordre 2');
hold on;
plot(t, y_rk4, 'b-*', 'DisplayName', 'Runge-Kutta Ordre 4');
legend;
xlabel('x');
ylabel('y');
title('Résolution des EDO par les méthodes de Runge-Kutta');
grid on;
```

Chapitre 5 : Systèmes d'équations linéaires

La résolution de systèmes d'équations linéaires constitue probablement le problème numérique le plus courant. Elle apparaît dans des domaines aussi variés que l'analyse des réseaux électriques en régime permanent, la résistance des matériaux, ou encore les modèles économétriques. Les problèmes aux limites, souvent associés à des équations différentielles ou aux dérivées partielles, sont fréquemment discrétisés, ce qui donne lieu à des systèmes d'équations linéaires. Ce domaine représente également l'un des plus étudiés et bien établis en analyse numérique, reposant sur les concepts fondamentaux de l'algèbre linéaire.

Par exemple, un système de $\boldsymbol{m}$ équations linéaires avec $\boldsymbol{n}$ variables peut être représenté de la manière suivante :

$$\begin{cases} \boldsymbol{a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1} \\ \boldsymbol{a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2} \\ \vdots \\ \boldsymbol{a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = b_m} \end{cases}$$

Eq. 9

Ce système peut s'écrire sous la forme matricielle suivante : $[A][X] = [b]$ avec, A est la matrice des coefficient, X est le vecteur des variables et b est le vecteur des constantes.

L'objectif est de trouver les valeurs des variables $\boldsymbol{x_1}$, $\boldsymbol{x_2}$, …, $\boldsymbol{x_n}$ qui satisfont toutes les équations simultanément. Cela peut être fait en utilisant différentes méthodes, telles que la méthode de Gauss-Jordan, Décomposition LU, Méthode de Jacobi, etc.

1 Méthode de Gauss [12]

1.1 Algorithme

L'objectif de l'algorithme de Gauss est de transformer la matrice A en une matrice triangulaire supérieure U (en utilisant des opérations élémentaires sur les lignes). Ces opérations incluent :

- Échanger deux lignes ;

- Multiplier une ligne par un scalaire non nul ;
- Ajouter un multiple d'une ligne à une autre.

Pour le système initial, nous utilisons une notation détaillée, soit

$$A^{(1)}x = b^{(1)}$$

Où les éléments de la matrice $A^{(1)}$ sont notés $a_{ij}^{(1)}$ et ceux de $b^{(1)}$ sont notés $b_i^{(1)}$. L'objectif est de transformer ce système en un système équivalent de la forme $Ux = c$, où U est une matrice triangulaire supérieure (U pour "upper"). Il est important de noter que la solution du système reste inchangée si :

- Une équation (une ligne de A et la coordonnée correspondante de b) est multipliée par une constante ;
- Une équation est ajoutée à une autre (une ligne de A à une autre et la coordonnée correspondante de b à l'autre).

Dans la première étape, nous éliminons x_1 de toutes les lignes, sauf de la première. Pour cela, nous supposons que le "pivot" $a_{11}^{(1)} \neq 0$ et définissons les "multiplicateurs" :

$$m_{i1} = -\frac{a_{i1}^{(1)}}{a_{11}^{(1)}}$$

Ensuite, nous obtenons une nouvelle ligne pour chaque ligne i en **ajoutant** à cette ligne m_{i1} fois la première ligne.

$$a_{ij}^{(2)} = a_{ij}^{(1)} + m_{i1}a_{1j}^{(1)} \qquad i, j = 2, \dots, n,$$

$$b_i^{(2)} = b_i^{(1)} + m_{i1}b_1^{(1)} \qquad i = 2, \dots, n,$$

Remarque : Beaucoup d'auteurs préfèrent adopter la convention inverse, où

$$m_{i1} = +a_{i1}^{(1)}/a_{11}^{(1)}$$

et soustraient m_{i1} fois la première ligne à la $i - ème$.

La première ligne de la matrice A et la première composante du vecteur b restent inchangées. Le système d'équations prend ainsi la forme suivante :

$$\begin{bmatrix} a_{i1}^{(1)} & a_{12}^{(1)} & \cdots & a_{1N}^{(1)} \\ 0 & a_{22}^{(2)} & \vdots & a_{2n}^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & a_{n2}^{(2)} & \cdots & a_{nn}^{(2)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1^{(1)} \\ b_2^{(2)} \\ \vdots \\ b_n^{(2)} \end{bmatrix}$$

On constate que l'inconnue x_1 a été éliminée des lignes 2 à n. Le cas général suit une structure similaire. Nous supposons avoir éliminé $x_1, x_2, \dots, x_{k-1}$, de sorte que la matrice des coefficients prend la forme :

$$\begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & & \cdots & & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & & \vdots & & a_{2n}^{(2)} \\ \vdots & & \ddots & & & \vdots \\ 0 & \cdots & 0 & a_{kk}^{(k)} & \cdots & a_{kn}^{(k)} \\ \vdots & & \vdots & & & \vdots \\ 0 & \cdots & 0 & a_{nk}^{(2)} & \cdots & a_{nn}^{(2)} \end{bmatrix}$$

Définissons un nouveau pivot $a_{kk}^{(k)}$, supposé différent de zéro, ainsi que les nouveaux multiplicateurs.

$$m_{ik} = -\frac{a_{ik}^{(k)}}{a_{kk}^{(k)}}, \qquad i = k+1, \dots, n$$

Eq. 10

En utilisant ces éléments, nous procédons à l'élimination de l'inconnue x_k des équations de rang $k+1, k+2, \dots, n$, ce qui donne les équations suivantes :

$$a_{ij}^{(k+1)} = a_{ij}^{(k)} + m_{ik} a_{kj}^{(k)}, \qquad i, j = k+1, \dots, n$$

$$b_i^{(k+1)} = b_i^{(k)} + m_{ik} b_k^{(k)}, \qquad i = k+1, \dots, n$$

Eq. 11

Nous laissons inchangées les lignes de rang 1 à k et remplaçons par des zéros les éléments de la colonne k situés sous le pivot. Après $n-1$ étapes similaires à celle décrite précédemment, l'élimination est terminée et le système prend la forme suivante

$$A^{(n)}x = b^{(n)}$$

Ou encore,

$$\begin{bmatrix} a_{i1}^{(1)} & a_{12}^{(1)} & \cdots & a_{1N}^{(1)} \\ 0 & a_{22}^{(2)} & \vdots & a_{2n}^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & a_{n2}^{(2)} & \cdots & a_{nn}^{(2)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1^{(1)} \\ b_2^{(2)} \\ \vdots \\ b_n^{(2)} \end{bmatrix}$$

Pour simplifier la notation, posons $U = A^{(n)}$ et $c = b^{(n)}$. Le système d'équations $\boldsymbol{U}x = c$ est alors sous forme triangulaire supérieure et peut être résolu facilement en procédant de bas en haut. Nous obtenons successivement :

$$x_n = \frac{c_n}{u_{nn}}$$

puis

$$x_k = \frac{1}{u_{kk}}\left[c_k - \sum_{j=k+1}^{n} u_{kj}x_j\right], \qquad k = n-1, n-2, \ldots, 1$$

1.2 Méthode de Gauss-Jordan

Pour ceux qui apprécient l'élimination, nous présentons une variante : la méthode de Gauss-Jordan. Plutôt que de s'arrêter à un système triangulaire, cette méthode poursuit l'élimination au-dessus de la diagonale principale. En sélectionnant un pivot et en définissant un multiplicateur selon l'équation (Eq. 10), les équations (Eq. 11) sont ensuite utilisées pour toutes les combinaisons $j = k, \ldots, n$ et $i = 1, \ldots, n$, avec $i \neq k$. À l'issue de ce processus, la matrice des coefficients devient diagonale, permettant de résoudre le système directement.

2 Décomposition de Crout et Factorisation LU [6]

2.1 Factorisation LU

Les multiplicateurs utilisés dans la méthode de Gauss vont désormais être exploités pour élaborer un algorithme qui, bien que présentant une apparence différente, est en réalité entièrement équivalent.

Définissons la matrice triangulaire inférieure (L pour « lower »)

$$\boldsymbol{L} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ -m_{21} & 1 & 0 & \cdots & 0 \\ \vdots & & \ddots & & \vdots \\ -m_{n1} & -m_{n2} & -m_{n3} & \cdots & 1 \end{bmatrix}$$

Le résultat remarquable qui en découle est que, si $\boldsymbol{L}$ et $\boldsymbol{U}$ sont obtenues par l'élimination de Gauss appliquée à la matrice $\boldsymbol{A}$, alors

$$\boldsymbol{A} = \boldsymbol{LU}$$

Pour le vérifier, considérons l'élément (i, j) du produit $\boldsymbol{LU}$. Cet élément correspond au produit scalaire du vecteur ligne $[-m_{i1}, -m_{i2}, \dots, -m_{i,i-1}, 1, 0, \dots, 0]$ de $\boldsymbol{L}$ avec le vecteur colonne $[u_{1j}, u_{2j}, \dots, u_{j,j}, 0, \dots, 0]^T$ de $\boldsymbol{U}$.

On distingue alors deux cas :

- $i \leq j$ (Au-dessus ou sur la diagonale principale),
- $i > j$ (En dessous de la diagonale principale).

Ces cas s'examinent en tenant compte des définitions des éléments de $\boldsymbol{L}$ et $\boldsymbol{U}$, notamment la première des équations (Eq. 11).

Si $i \leq j$:

$$(LU)_{ij} = -\sum_{k=1}^{i-1} m_{ik}u_{kj} + u_{ij} = -\sum_{k=1}^{i-1} m_{ik}a_{kj}^{(k)} + a_{ij}^{(i)} = -\sum_{k=1}^{i-1}\left[a_{ij}^{(k+1)} - a_{ij}^{(k)}\right] + a_{ij}^{(i)} = a_{ij}^{(1)} = a_{ij}$$

Si $i > j$:

$$(LU)_{ij} = -\sum_{k=1}^{j} m_{ik}u_{kj} = -\sum_{k=1}^{j-1} m_{ik}a_{kj}^{(k)} + a_{ij}^{(j)} = -\sum_{k=1}^{j-1}\left[a_{ij}^{(k+1)} - a_{ij}^{(k)}\right] + a_{ij}^{(j)} = a_{ij}^{(1)} = a_{ij}$$

Le déterminant d'un produit de matrices étant égal au produit des déterminants de chaque facteur, et le déterminant d'une matrice triangulaire étant donné par le produit de ses éléments diagonaux, la factorisation LU permet également de calculer directement le déterminant de $\boldsymbol{A}$:

$$\det(\boldsymbol{A}) = \det(\boldsymbol{L})\det(\boldsymbol{U}) = u_{11}u_{22}\ldots u_{nn}$$

Pour résoudre le système (Eq. 9), une fois les matrices $\boldsymbol{L}$ et $\boldsymbol{U}$ obtenues, il suffit de résoudre deux systèmes triangulaires successivement.

Résolvons d'abord :

$$\boldsymbol{Ly} = \boldsymbol{b}$$

Eq. 12

afin d'obtenir le vecteur $\boldsymbol{y}$.

Puis :

$$\boldsymbol{Ux} = \boldsymbol{y}$$

Eq. 13

dont la solution est le vecteur $\boldsymbol{x}$ cherché. En effet, en substituant $\boldsymbol{y}$ tiré de (Eq. 13) dans (Eq. 12), nous trouvons $\boldsymbol{LUx} = \boldsymbol{Ax} = \boldsymbol{b}$. Cette démarche est profitable si l'on doit résoudre plusieurs

systèmes d'équations linéaires qui ne diffèrent que par leurs seconds membres : la factorisation est faite une fois pour toutes.

2.2 Décomposition de Crout

Une matrice régulière $\boldsymbol{A}$ peut être exprimée comme le produit de deux matrices triangulaires $\boldsymbol{L}$ et $\boldsymbol{U}$. Cela soulève une question intéressante : est-il possible de déterminer les éléments de $\boldsymbol{L}$ et $\boldsymbol{U}$ sans recourir à l'algorithme de Gauss ? La réponse est oui : il suffit d'identifier chaque élément en utilisant l'équation $\boldsymbol{A} = \boldsymbol{LU}$. Étant donné que la décomposition $\boldsymbol{LU}$ est unique, cet algorithme alternatif est équivalent à celui de Gauss.

Soient l_{ij} les éléments de L, avec $l_{ij} = 0$ si $i < j$; de même, les éléments de $\boldsymbol{U}$ sont tels que $u_{ij} = 0$ si $i > j$; enfin $l_{ii} = 0$. Les éléments de la première ligne de $\boldsymbol{A}$ vérifient

$$a_{11} = \sum l_{1k}u_{k1} = l_{11}u_{11} = u_{11}$$
$$a_{12} = \sum l_{1k}u_{k2} = u_{12}$$
$$\dots$$
$$a_{1n} = \sum l_{1k}u_{kn} = u_{1n}$$

En utilisant la première ligne de $\boldsymbol{A}$, nous avons trouvé la première ligne de $\boldsymbol{U}$. L'astuce pour continuer (due à ***Crout***, d'où le nom de l'algorithme), consiste à identifier maintenant la première colonne de $\boldsymbol{A}$.

$$a_{21} = \sum l_{2k}u_{k1} = l_{21}u_{11} + l_{22}u_{21} \quad \Rightarrow \quad l_{21} = a_{21}/u_{11}$$
$$a_{31} = \sum l_{3k}u_{k1} = l_{31}u_{11} + l_{32}u_{21} + l_{33}u_{31} \quad \Rightarrow \quad l_{31} = a_{31}/u_{11}$$
$$\dots$$

Nous déterminons ainsi la première colonne de $\boldsymbol{L}$. Vous pouvez imaginer que la deuxième ligne de $\boldsymbol{U}$ s'obtient à partir de la deuxième ligne de $\boldsymbol{A}$, puis que la deuxième colonne de $\boldsymbol{L}$ découle de la deuxième colonne de $\boldsymbol{A}$, etc. Les formules générales s'écrivent :

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj}\,; \qquad j = i, i+1, \dots, n;$$

$$l_{ij} = \frac{1}{u_{ij}}\left[a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj}\right]; \qquad i = j+1, j+2, \dots, n,$$

avec toujours $l_{ii} = 1$. Le rôle de pivot est tenu par l'élément u_{ij}, qui doit être non-nul. Une condition équivalente est que $det(A(1{:}\,k, 1{:}\,k)) \neq 0$ pour $1 \leq k \leq n-1$; autrement dit, toutes les sous-matrices que l'on peut extraire du coin supérieur gauche de $\boldsymbol{A}$ doivent être régulières. En pratique, non seulement le pivot ne doit pas être nul mais, comme cet algorithme est sensible aux erreurs d'arrondis, il faut choisir le plus grand élément de la colonne *j* comme pivot et effectuer les permutations de lignes correspondantes.

3 Méthodes itératives de résolution des systèmes linéaires

Les systèmes linéaires de très grande taille présentent une difficulté pratique : il peut être impossible de stocker tous les coefficients en mémoire centrale. De plus, ces systèmes, notamment lorsqu'ils proviennent de la discrétisation d'une équation aux dérivées partielles, présentent souvent une structure particulière des coefficients, avec des éléments diagonaux nettement supérieurs aux autres. Dans de tels cas, les méthodes itératives de résolution offrent une solution avantageuse, étant à la fois moins exigeantes en mémoire et plus rapides.

3.1 Méthode de Jacobi

Nous cherchons à résoudre l'équation (Eq. 9), avec $a_{ii} \neq 0$ pour $\boldsymbol{1 \leq i \leq n}$, en partant d'une approximation initiale de la solution, $x^{(0)}$. Une méthode d'approximations successives est utilisée, définie par les relations suivantes :

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j \neq i} a_{ij}x_j^{(k)}\right); \qquad \boldsymbol{1 \leq i \leq n}$$

Eq. 14

Le calcul de $x_i^{(k+1)}$ s'appuie uniquement sur la ligne $\boldsymbol{i}$ de la matrice $\boldsymbol{A}$, ce qui permet de stocker cette matrice sur disque et de la lire ligne par ligne, à la demande. Il est important de noter que les valeurs les plus récentes ne sont pas utilisées lors du calcul de $x_i^{(k+1)}$.

Comme pour toute méthode itérative, un critère d'arrêt est nécessaire. Celui-ci peut consister à surveiller la variation absolue ou relative de la norme de $x_i^{(k)}$ d'une itération à l'autre. Une condition d'arrêt relative peut alors s'écrire :

$$\left\|x^{(k+1)} - x^{(k)}\right\| \leq \varepsilon \left\|x^{(k)}\right\|$$

où ε est un seuil prédéfini.

Un autre critère repose sur le calcul du résidu à l'itération k, défini comme :

$$\boldsymbol{r^{(k)} \equiv b - Ax^{(k)}}$$

La condition de convergence peut ensuite être exprimée à partir de ce résidu :

$$\left\|\boldsymbol{r^{(k)}}\right\| \leq \boldsymbol{\eta}\|\boldsymbol{b}\|$$

Soit $\boldsymbol{\eta}$ un seuil prédéfini

L'algorithme défini par les équations (Eq. 14) peut être exprimé de manière matricielle, ce qui facilite les analyses théoriques. Pour cela, décomposons la matrice $\boldsymbol{A}$ en la somme de trois matrices distinctes (sans lien avec les matrices $\boldsymbol{L}$ et $\boldsymbol{U}$ évoquées précédemment) :

$$\boldsymbol{A = D - L - U}$$

où $\boldsymbol{D}$ est une matrice diagonale, $\boldsymbol{L}$ est strictement triangulaire inférieure, et $\boldsymbol{U}$ est strictement triangulaire supérieure. Par conséquent, les éléments diagonaux de $\boldsymbol{A}$ se retrouvent dans $\boldsymbol{D}$, et les éléments diagonaux de $\boldsymbol{L}$ et $\boldsymbol{U}$ sont nuls ($a_{ii} = d_{ii}, l_{ii} = u_{ii} = 0, 1 \leq i \leq n$)

Comme $\boldsymbol{D}$ est diagonale, son inverse l'est également, avec des éléments donnés par $[D^{-1}]_{ii} = 1/d_{ii} = 1/a_{ii}$. Les équations (Eq. 14) peuvent alors être reformulées de manière compacte :

$$x^{(k+1)} = D^{-1}(L + U)x^{(k)} + D^{-1}b$$

3.2 Méthode Gauss-Seidel

L'algorithme de Jacobi nous prescrit de calculer $x_i^{(k+1)}$ en utilisant les valeurs $x_1^{(k)}, x_2^{(k)}, \dots, x_{i-1}^{(k)}, x_{i+1}^{(k)}, \dots, x_n^{(k)}$; il semble que nous pourrions gagner du temps ou de la précision, puisque nous connaissons déjà $x_1^{(k+1)}, x_2^{(k+1)}, \dots, x_{i-1}^{(k+1)}$, qui sont plus précises que les valeurs des mêmes variables à l'itération précédente k. C'est ce que fait l'algorithme de Gauss–Seidel. Les relations de récurrence s'écrivent :

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right)$$

Eq. 15

Pour découvrir l'équivalent matriciel des équations (Eq. 15), il suffit de les « déplier » pour les mettre sous la forme équivalente

$$\sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} + a_{ii} x_i^{(k+1)} = b_i - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)}$$

Le membre de gauche représente la ligne i du produit $(-L + D)x^{(k+1)}$, tandis que le membre de droite est la ligne i de $b + Ux^{(k)}$. Le passage de $x^{(k)}$ à $x^{(k+1)}$ peut donc s'écrire

$$(D - L)x^{(k+1)} = Ux^{(k)} + b$$

4 TP5 : Résolution des systèmes d'équations linéaires

Objectif du TP : Programmation des méthodes, Gauss-Jordan, LU, Jacobi et Gauss-Seidel pour la résolution d'un système d'équations linéaires.

Système d'équations linéaires :

Un système d'équations linéaires est un ensemble d'équations linéaires impliquant plusieurs variables. Par exemple, un système de $\underline{\boldsymbol{m}}$ équations linéaires avec $\underline{\boldsymbol{n}}$ variables peut être représenté de la manière suivante :

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \quad\quad\quad\quad . \\ \quad\quad\quad\quad . \\ \quad\quad\quad\quad . \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = b_m \end{cases}$$

Ce système peut s'écrire sous la forme matricielle suivante : **[A][X] = [b]** avec, **A** est la matrice des coefficient, **X** est le vecteur des variables et **b** est le vecteur des constantes.

L'objectif est de trouver les valeurs des variables $\boldsymbol{x_1, x_2, \ldots, x_n}$ qui satisfont toutes les équations simultanément. Cela peut être fait en utilisant différentes méthodes, telles que la méthode de Gauss-Jordan, Décomposition LU, Méthode de Jacobi, etc.

Travail demandé

Soit le système d'équations linéaires suivant :

$$\begin{gathered} 2x_1 + x_2 - x_3 = 5 \\ x_1 - x_2 + 3x_3 = 10 \\ 3x_1 + 2x_2 - x_3 = 3 \end{gathered}$$

Ecrire l'algorithme qui permet de trouver la solution de ce système en utilisant les méthodes (Gauss, LU, Jacobi et Gauss-seidel)

Ecrire le programme MATLAB correspondant à chaque méthode.

Dérouler manuellement les programmes.

Solution

Exemple de script MATLAB de l'algorithme de Gauss

```
function x = gaussElimination(A, b)
    % Résolution d'un système linéaire Ax = b par l'algorithme de
Gauss.
    % Entrées :
    %   A : matrice des coefficients (n x n)
    %   b : vecteur des termes constants (n x 1)
    % Sortie :
    %   x : solution du système (n x 1)

    [n, m] = size(A);
    if n ~= m
        error('La matrice A doit être carrée.');
    end

    if length(b) ~= n
        error('La dimension du vecteur b doit correspondre à celle
de A.');
    end

    % Augmentation de la matrice A avec le vecteur b
    Ab = [A, b];

    % Étape d'élimination
    for k = 1:n-1
        % Pivot partiel (optionnel pour améliorer la stabilité)
        [~, maxIndex] = max(abs(Ab(k:n, k)));
        maxIndex = maxIndex + k - 1;
        if maxIndex ~= k
```

```
            Ab([k, maxIndex], :) = Ab([maxIndex, k], :);
        end

        % Élimination des termes sous le pivot
        for i = k+1:n
            factor = Ab(i, k) / Ab(k, k);
            Ab(i, k:end) = Ab(i, k:end) - factor * Ab(k, k:end);
        end
    end

    % Vérification si le système est bien défini
    if any(diag(Ab(:, 1:n)) == 0)
        error('Le système est soit indéterminé, soit
incompatible.');
    end

    % Substitution arrière pour obtenir la solution
    x = zeros(n, 1);
    for i = n:-1:1
        x(i) = (Ab(i, end) - Ab(i, i+1:n) * x(i+1:n)) / Ab(i, i);
    end
end

% Exemple d'utilisation
A = [2, -1, 1; 1, 3, 2; 1, -1, 2];
b = [1; 12; 3];
x = gaussElimination(A, b);
disp('Solution :');
disp(x);
```

Exemple d'un script MATLAB de l'algorithme LU

```
A = [2 1 -1; 1 -1 3; 3 2 -1];          % Définition de la matrice A
b = [5; 10; 3];                        % Définition du vecteur b
[n, ~] = size(A);                      % Obtenir la taille de la
matrice A
% Initialiser les matrices L et U
L = eye(n);                            % L commence comme matrice
identité
U = A;                                 % U commence comme la matrice
A
% Décomposition LU
for j = 1:n-1
    for i = j+1:n
        L(i, j) = U(i, j) / U(j, j); % Calculez le coefficient de
mise à l'échelle
        U(i, :) = U(i, :) - L(i, j) * U(j, :); % Mettre à jour les
lignes
    end
end
% Étape 1 : Résoudre Ly = b avec substitution vers l'avant
y = zeros(n, 1);                       % Initialiser y
for i = 1:n
    y(i) = b(i);
    for j = 1:i-1
        y(i) = y(i) - L(i, j) * y(j);
    end
end
% Étape 2 : Résoudre Ux = y avec substitution vers l'arrière
x = zeros(n, 1);                       % Initialiser x
for i = n:-1:1
    x(i) = y(i);
    for j = i+1:n
```

```
        x(i) = x(i) - U(i, j) * x(j);
    end
    x(i) = x(i) / U(i, i);
end
disp('La solution x est:')                % Afficher la solution x
disp(x)
```

Script MATLAB de l'algorithme de JACOBI

```
% Script MATLAB pour la méthode de Jacobi avec décomposition D, L,
U

% Définir la matrice A et le vecteur b
A = [10 -1 2 0;
     -1 11 -1 3;
     2 -1 10 -1;
     0 3 -1 8]; % Exemple de matrice
b = [6; 25; -11; 15]; % Exemple de vecteur

% Vérification de la diagonale dominante (optionnel mais
recommandé)
if any(2 * abs(diag(A)) <= sum(abs(A), 2))
    error('La matrice A doit être strictement diagonale dominante
pour garantir la convergence.');
end

% Décomposer A en D, L et U
D = diag(diag(A));          % Matrice diagonale
L = tril(A, -1);            % Matrice strictement triangulaire
inférieure
U = triu(A, 1);             % Matrice strictement triangulaire
supérieure
```

```
% Initialisation
n = length(b);              % Nombre d'inconnues
x = zeros(n, 1);            % Approximation initiale
tol = 1e-6;                 % Tolérance pour le critère d'arrêt
max_iter = 100;             % Nombre maximum d'itérations
iter = 0;                   % Compteur d'itérations

% Pré-calcul pour simplifier les itérations
D_inv = inv(D);             % Calcul de D^(-1) (diagonale facile à
inverser)
T = -D_inv * (L + U);       % Matrice d'itération
c = D_inv * b;              % Terme constant

% Méthode de Jacobi
fprintf('Début des itérations de Jacobi...\n');
while iter < max_iter
    iter = iter + 1;
    x_new = T * x + c;      % Mise à jour selon x^(k+1) = T*x^(k) +
c

    % Vérification du critère d'arrêt
    if norm(x_new - x, inf) < tol
        fprintf('Convergence atteinte après %d itérations.\n',
iter);
        break;
    end

    % Mise à jour pour l'itération suivante
    x = x_new;
end
```

```
% Résultats
if iter == max_iter
    fprintf('Nombre maximum d''itérations atteint sans
convergence.\n');
else
    fprintf('Solution approchée :\n');
    disp(x_new);
end

% Affichage des matrices D, L, U
fprintf('Matrice D :\n');
disp(D);
fprintf('Matrice L :\n');
disp(L);
fprintf('Matrice U :\n');
disp(U);
```

Script MATLAB de l'algorithme de Gauss-Seidel

```
% Script MATLAB pour la méthode de Gauss-Seidel avec décomposition
D, L, U

% Définir la matrice A et le vecteur b
A = [10 -1 2 0;
     -1 11 -1 3;
     2 -1 10 -1;
     0 3 -1 8]; % Exemple de matrice
b = [6; 25; -11; 15]; % Exemple de vecteur

% Vérification de la diagonale dominante (optionnel mais
recommandé)
```

```
if any(2 * abs(diag(A)) <= sum(abs(A), 2))
    error('La matrice A doit être strictement diagonale dominante
pour garantir la convergence.');
end

% Décomposer A en D, L et U
D = diag(diag(A));          % Matrice diagonale
L = tril(A, -1);            % Matrice strictement triangulaire
inférieure
U = triu(A, 1);             % Matrice strictement triangulaire
supérieure

% Initialisation
n = length(b);              % Nombre d'inconnues
x = zeros(n, 1);            % Approximation initiale
tol = 1e-6;                 % Tolérance pour le critère d'arrêt
max_iter = 100;             % Nombre maximum d'itérations
iter = 0;                   % Compteur d'itérations

% Pré-calcul pour simplifier les itérations
T = -(D + L) \ U;           % Matrice d'itération
c = (D + L) \ b;            % Terme constant

% Méthode de Gauss-Seidel
fprintf('Début des itérations de Gauss-Seidel...\n');
while iter < max_iter
    iter = iter + 1;
    x_new = T * x + c;      % Mise à jour selon x^(k+1) = T*x^(k) +
c

    % Vérification du critère d'arrêt
```

```
    if norm(x_new - x, inf) < tol
        fprintf('Convergence atteinte après %d itérations.\n',
iter);
        break;
    end

    % Mise à jour pour l'itération suivante
    x = x_new;
end

% Résultats
if iter == max_iter
    fprintf('Nombre maximum d''itérations atteint sans
convergence.\n');
else
    fprintf('Solution approchée :\n');
    disp(x_new);
end

% Affichage des matrices D, L, U
fprintf('Matrice D :\n');
disp(D);
fprintf('Matrice L :\n');
disp(L);
fprintf('Matrice U :\n');
disp(U);
```

Bibliographie

[1] J. Ouin, Algorithmique et calcul numérique : travaux pratiques résolus et programmation avec les logiciels Scilab et Python, Paris: Ellipses, 2013.

[2] J.-P. Grenier, Débuter en algorithmique avec MATLAB et SCILAB, Paris: Ellipses, 2007.

[3] M. Djebli et H. Djelouah, Initiation à matlab, Alger: Office des publications universitaires, 2016.

[4] M.-H. Meurisse, Algorithmes numériques : fondements théoriques et analyse pratique : cours, exercices et applications avec MATLAB, Paris: Ellipses, 2018.

[5] A. Quarteroni, S. Riccardo et S. Fausto, Méthodes numériques pour le calcul scientifique : programmes en Matlab, Paris: Springer, 2000.

[6] J.-P. Grivet, Méthodes numériques appliquées : pour le scientifique et l'ingénieur, Paris: EDP sciences, 2009.

[7] R. Dualé, Méthodes numériques, Paris: Techniques de l'ingénieur, 2012.

[8] P. Destuynder, Méthodes numériques pour l'ingénieur, Paris: Hermès science publications-Lavoisier, 2010.

[9] F. Jedrzejewski, Introduction aux méthodes numériques, Paris: Springer, 2005.

[10] G. Jean-Baptiste, Approche fonctionnelle des calculs scientifiques : méthodes numériques et applications : langage Python, Toulouse: Cepadues-Editions, 2016.

[11] A. E. H. Bouchaib Radi, Mathématiques avec Scilab : guide de calcul programmation représentations graphiques : conforme au nouveau programme MPSI, Paris: Ellipses, 2009.

[12] B. Michaël, Méthodes numériques avec Python : théorie, algorithmes, implémentation et applications avec Python 3, Malakoff : Dunod, 2023.