Université Amine Elokkal El Hadj Moussa de Tamanrasset

Département Mathématique et Informatique

POLYCOPIE DE COURS

Outils de Programmation pour les mathématiques

Auteur :
Dr. Thiziri SIFAOUI

Examinateurs :
Pr. Ali LEMOUARI
Dr. Ammar ABDELLI

Année académique 2023/2024

Université Amine Elokkal El Hadj Moussa de Tamanrasset

Département Mathématique et Informatique

POLYCOPIE DE COURS

Outils de Programmation pour les mathématiques

Concepteur de polycopie :
Dr. Thiziri SIFAOUI

Examinateurs :
Pr. Ali LEMOUARI
Dr. Ammar ABDELLI

Année académique 2023/2024

Table des matières

Introduction générale

Scilab est un logiciel de calcul numérique et de programmation scientifique, largement utilisé dans les domaines de l'ingénierie, des sciences et des mathématiques. Il offre un environnement de développement convivial et puissant pour résoudre une grande variété de problèmes mathématiques et scientifiques.

Scilab est un logiciel open source, ce qui signifie qu'il est gratuit et que son code source est accessible à tous. Il a été développé à l'origine par l'INRIA (Institut National de Recherche en Informatique et en Automatique) en France, et est maintenant maintenu par une communauté internationale de développeurs.

Scilab est basé sur le langage de programmation Scilab, qui est similaire à Matlab. Il offre une large gamme de fonctionnalités pour effectuer des calculs numériques, des analyses statistiques, des simulations, des visualisations de données, et bien plus encore. Il est utilisé par des chercheurs, des ingénieurs, des étudiants et des professionnels du monde entier pour résoudre des problèmes complexes et réaliser des analyses avancées.

L'une des principales caractéristiques de Scilab est sa capacité à manipuler des tableaux et des matrices de manière efficace. Il offre des fonctions puissantes pour effectuer des opérations mathématiques sur ces structures de données, ce qui en fait un outil idéal pour la manipulation de données scientifiques et l'analyse numérique.

Scilab dispose également d'une interface utilisateur conviviale qui facilite l'interaction avec le logiciel. L'interface utilisateur de Scilab comprend une fenêtre de commande où vous pouvez saisir des commandes et exécuter des scripts, ainsi qu'une fenêtre de graphiques pour afficher les résultats des calculs et des visualisations.

L'installation de Scilab est relativement simple et peut être effectuée sur différentes plateformes, y compris Windows, macOS et Linux. Une fois installé, vous pouvez commencer à utiliser Scilab en ouvrant simplement l'application et en commençant à saisir des commandes dans la fenêtre de commande.

Scilab est également extensible, ce qui signifie que vous pouvez ajouter des fonctionnalités supplémentaires en utilisant des modules et des boîtes à outils. Ces modules et boîtes à outils fournissent des fonctionnalités supplémentaires pour des domaines spécifiques tels que l'optimisation, le traitement du signal, la modélisation et la simulation, et bien d'autres encore.

En résumé, Scilab est un logiciel puissant et polyvalent pour le calcul numérique et la programmation scientifique. Il offre une large gamme de fonctionna-

lités pour résoudre des problèmes mathématiques et scientifiques, et est utilisé par des chercheurs, des ingénieurs et des étudiants du monde entier. Que vous soyez un débutant ou un utilisateur expérimenté, Scilab peut vous aider à réaliser des analyses avancées et à résoudre des problèmes complexes de manière efficace et précise.

Chapitre 1

Prise en Main

1.1 Installation :

Scilab, logiciel de calcul numérique gratuit et open-source, représente une alternative prisée à MATLAB, notamment dans le cadre éducatif, du lycée jusqu'à l'université. Sa polyvalence en fait un outil apprécié pour diverses tâches liées au calcul numérique, à l'analyse de données et à la modélisation mathématique. Son interface conviviale et ses similitudes avec MATLAB au niveau du langage de programmation en font un choix attractif pour les étudiants et les professionnels. De plus, la communauté active qui l'entoure assure un support continu et offre une variété de ressources pour faciliter son apprentissage et son utilisation.

Scilab est disponible pour les systèmes d'exploitation Windows, Linux et macOS, offrant ainsi une accessibilité étendue à ses utilisateurs. Vous pouvez le télécharger à partir du site officiel du logiciel Scilab.

Une fois lancé, le programme dévoile une interface de travail conviviale, prête à répondre à vos besoins en calcul numérique, en analyse de données et en modélisation mathématique.

1.2 Présentation IDE :

Scilab propose son propre environnement de développement intégré (IDE) conçu pour faciliter le travail dans son logiciel de calcul numérique. Cet IDE est une plateforme centralisée regroupant divers outils pour la modélisation mathématique, la simulation, l'analyse de données et d'autres tâches liées au calcul scientifique. Il offre une interface conviviale pour l'écriture de code, la visualisation des résultats et la gestion de projets.

Cet environnement de développement fournit un éditeur de code avec des fonctionnalités telles que la coloration syntaxique, l'autocomplétion, et d'autres outils d'aide à la programmation. De plus, il intègre des outils avancés tels que

des modules pour le traitement du signal, la simulation de systèmes dynamiques, ainsi que des fonctionnalités pour le traitement de données et l'optimisation.

L'IDE de Scilab offre également la possibilité de créer, gérer et organiser des projets, simplifiant ainsi le flux de travail pour les utilisateurs. Il peut être personnalisé pour répondre aux besoins spécifiques des utilisateurs et offre une expérience utilisateur fluide pour les professionnels et les étudiants travaillant dans des domaines tels que les sciences, l'ingénierie et la recherche.

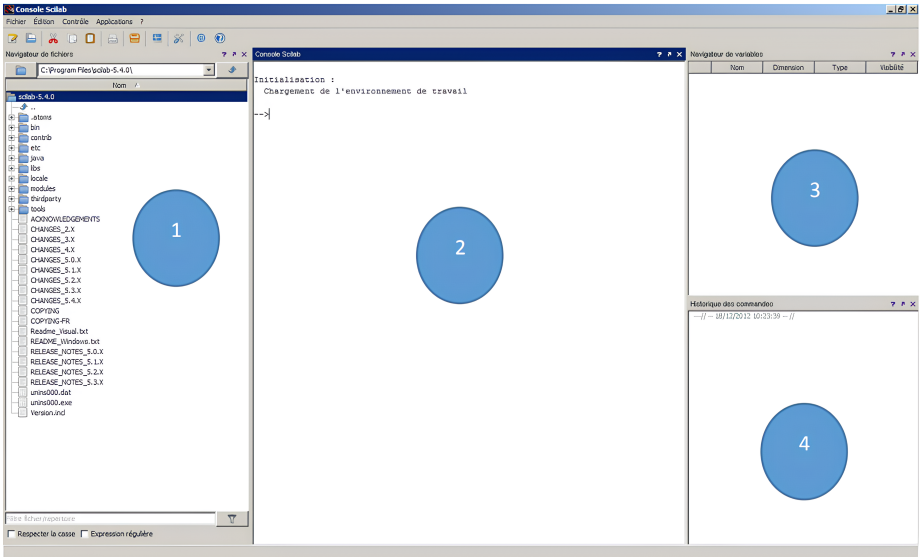

FIGURE 1.1 – L'IDE de Scilab.

1 : La zone de chargement du répertoire de travail et des fichiers fait référence à l'endroit où vous pouvez naviguer dans les répertoires de votre système de fichiers pour sélectionner le dossier ou les fichiers sur lesquels vous voulez travailler.

2 : La zone de commande ou le terminal est un espace dans un logiciel ou un environnement où vous saisissez des commandes ou des instructions spécifiques pour effectuer des opérations, lancer des fonctions ou afficher des résultats.

3 : La zone des variables ou des tableaux dans un environnement de programmation ou dans un logiciel de calcul est un espace dédié à l'affichage et à la gestion des données stockées sous forme de variables ou de tableaux.

4 : La zone de l'historique des commandes fait référence à une fonctionnalité dans un terminal ou une console où les commandes précédemment exécutées sont enregistrées et conservées pour référence ultérieure.

Pour effacer l'historique des commandes dans une console ou un terminal :

— **Méthode 1 :** Effacer depuis la console ou le terminal : En général, vous pouvez effacer l'écran de la console en utilisant la commande **clc** dans la plupart des terminaux.

— **Méthode 2 :** Effacer l'historique depuis l'interface de la console :

  — Dans certains terminaux ou consoles, vous pouvez faire un clic droit sur la zone de l'historique (là où s'affichent les commandes précédentes).

  — Sélectionnez l'option "Effacer l'historique" ou une option similaire dans le menu contextuel qui s'ouvre après le clic droit.

En Scilab, pour interrompre un programme en cours d'exécution, vous pouvez utiliser quelques méthodes :

1 : Utilisation du Clavier : **Ctrl + C :** Dans la console de Scilab, l'utilisation de la combinaison de touches **Ctrl + C** peut souvent interrompre l'exécution d'un script ou d'une série d'instructions en cours.

2 : Utilisation de la Fonction Interrupt : Scilab propose une fonction nommée **interrupt()** qui peut être utilisée pour interrompre l'exécution d'une tâche en cours.

3 : Utilisation de la Barre de Contrôle : Dans l'environnement de Scilab, il y a souvent des options disponibles dans la barre de contrôle ou dans les menus qui permettent d'arrêter l'exécution d'un programme.

Ces méthodes peuvent varier en fonction de la version de Scilab ou de l'environnement spécifique que vous utilisez. Il est recommandé de consulter la documentation de Scilab ou l'aide intégrée pour des informations précises sur la manière d'interrompre l'exécution d'un programme dans votre contexte particulier.

1.3 Éditeur :

Utiliser la console présente deux inconvénients majeurs : la possibilité d'enregistrer le code tapé est absente et la modification de plusieurs lignes d'instructions devient difficile. Pour accéder à l'éditeur depuis la console, vous pouvez simplement cliquer sur l'icône correspondante dans la barre d'outils ou choisir **Applications** > **SciNotes** dans la barre de menus. Une autre alternative consiste à saisir simplement **edit** dans la console. L'éditeur s'ouvre alors avec un fichier par défaut nommé 'sans titre 1', prêt à recevoir un nouveau script.

```
vecteur = [1, 2, 3, 4, 5];

somme = 0;
for i = 1:length(vecteur)
    somme = somme + vecteur(i);
end

disp("Vecteur: ");
disp(vecteur);
disp("Somme des elements: ");
disp(somme);
```

1.4 Pour exécuter :

L'exécution de Scilab offre un cadre puissant pour le développement et l'exécution de programmes scientifiques et d'algorithmes. En cliquant sur l'option "Exécuter" dans la barre de menus, plusieurs choix s'offrent pour l'exécution du code. L'option "fichier sans écho" permet une exécution silencieuse, idéale pour tester des scripts sans afficher le déroulement du programme dans la console. À l'inverse, "fichier avec écho" réécrit et exécute le code, offrant une visibilité étape par étape du programme pendant son exécution. Pour des tests plus ciblés, "jusqu'au curseur, avec écho" permet de sélectionner une partie précise du code pour une exécution détaillée dans la console. Cette variété d'options confère une grande flexibilité lors du développement, permettant un contrôle précis sur la manière dont le code est exécuté et affiché, facilitant ainsi le débogage, la compréhension et l'optimisation des programmes en Scilab.

1.5 La fenêtre Graphique :

Pour visualiser des représentations graphiques telles que des courbes et des graphiques, vous utilisez une fenêtre dédiée à cet effet. Cette interface graphique permet la création et la manipulation de différentes représentations visuelles. Pour générer un exemple de courbe, vous pouvez saisir dans la console la commande **plot**.

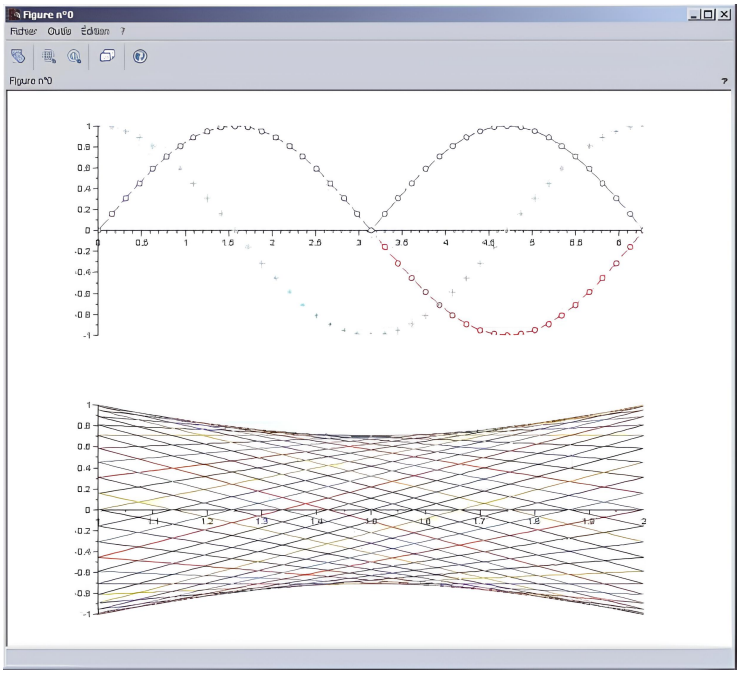

FIGURE 1.2 – La fenêtre Graphique de Scilab.

1.6 Installation toolbox :

Pour installer une toolbox (boîte à outils) dans Scilab, plusieurs méthodes peuvent être utilisées en fonction de la source de la toolbox :

1.6.1 À partir de l'interface utilisateur de Scilab :

1 : Via l'outil ATOMS :

- Dans Scilab, ouvrez la fenêtre de l'ATOMS Manager en sélectionnant "Applications" dans la barre de menu, puis "ATOMS".
- Recherchez la toolbox que vous souhaitez installer dans la liste proposée.
- Sélectionnez la toolbox et cliquez sur le bouton "Installer". Scilab téléchargera et installera automatiquement la toolbox.

2 : À partir d'un fichier de toolbox :

- Si vous avez le fichier de la toolbox (généralement un fichier .zip ou .tzb), vous pouvez l'installer en ouvrant l'ATOMS Manager.
- Cliquez sur "Install a new toolbox (offline)".
- Sélectionnez le fichier de la toolbox depuis votre ordinateur pour l'installer dans Scilab.

1.6.2 En ligne de commande (dans la console Scilab) :

Utiliser la fonction `atomsInstall()` pour installer une toolbox directement depuis la console en spécifiant le nom de la toolbox à installer :

```
atomsInstall('nom de la toolbox')
```

Conseil

Assurez-vous d'être connecté à Internet pour pouvoir utiliser l'ATOMS Manager ou la fonction `atomsInstall()`. Cela vous permettra de télécharger et d'installer la toolbox directement depuis le référentiel en ligne de Scilab.

1.7 Documentation :

La documentation en Scilab peut être obtenue via la commande `help` ou en accédant à la documentation en ligne officielle.

Pour obtenir de l'aide sur une fonction spécifique dans Scilab, utilisez `help` suivi du nom de la fonction. Par exemple :

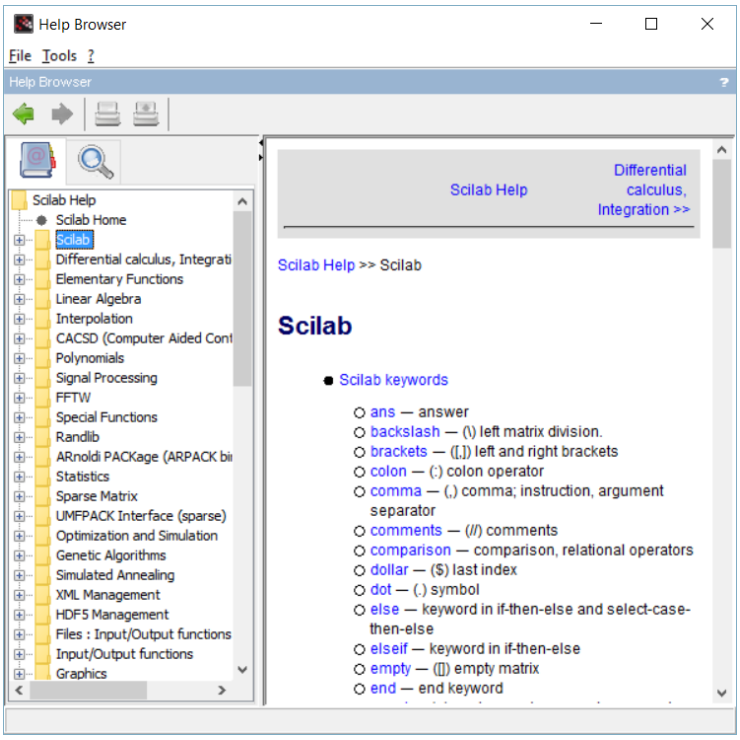

FIGURE 1.3 – La fenêtre Help de Scilab.

Travaux Pratique 1 : L'installation de Scilab

Installation de Base

— **Objectif :** Installer Scilab sur votre ordinateur.

— **Instructions :** Suivez le guide d'installation fourni pour installer Scilab sur votre système d'exploitation. Assurez-vous que Scilab fonctionne correctement après l'installation en le lançant et en explorant son interface.

L'ATOMS Manager

— **Objectif :** Utiliser l'ATOMS Manager pour installer une toolbox.

— **Instructions :** Ouvrez Scilab et accédez à l'ATOMS Manager. Choisissez une toolbox à installer et suivez les étapes pour l'installer via l'ATOMS Manager. Vérifiez que la toolbox est installée et explorez ses fonctionnalités.

Installation Hors Ligne

— **Objectif :** Installer une toolbox depuis un fichier.

— **Instructions :** Téléchargez le fichier de toolbox fourni (au format .zip ou .tzb). Utilisez l'option "Installer une nouvelle toolbox (hors ligne)" de l'ATOMS Manager pour installer cette toolbox depuis votre ordinateur. Vérifiez son installation et testez quelques fonctionnalités.

Installation via la Ligne de Commande

— **Objectif :** Installer une toolbox en utilisant une commande.

— **Instructions :** Utilisez la fonction `atomsInstall()` dans la console Scilab pour installer une toolbox spécifique en utilisant son nom. Vérifiez que la toolbox est correctement installée et explorez son contenu.

Correction de TP 1 : L'installation de Scilab

Installation de Base

Instructions :

— **Guide d'Installation :** Téléchargez le guide d'installation de Scilab à partir du site officiel www.scilab.org ou utilisez les instructions fournies par votre instructeur.

— **Installation de Scilab :** Suivez attentivement les étapes du guide pour installer Scilab sur votre système d'exploitation (Windows, macOS, Linux). Veillez à sélectionner les options appropriées pendant le processus d'installation.

— **Vérification de l'Installation :** Après avoir terminé l'installation, lancez Scilab depuis le raccourci sur votre bureau ou depuis le dossier d'installation. Explorez l'interface utilisateur pour vous familiariser avec ses fonctionnalités de base.

— **Vérification des Fonctionnalités :** Testez quelques fonctionnalités de Scilab, telles que la création de variables, l'exécution de commandes simples et l'affichage de graphiques simples.

Conseil

Si vous rencontrez des problèmes lors de l'installation ou si quelque chose ne fonctionne pas comme prévu, référez-vous au guide d'installation ou demandez de l'aide à votre instructeur.

l'ATOMS Manager

Instructions :

— **Ouvrir Scilab :** Lancez Scilab sur votre ordinateur.

— **Accès à l'ATOMS Manager :** Dans la barre de menu de Scilab, cliquez sur "Applications" puis sélectionnez "ATOMS". Cela ouvrira l'ATOMS Manager, une interface permettant la gestion et l'installation de toolboxes.

— **Choisir une Toolbox :** Parcourez la liste des toolboxes disponibles dans l'ATOMS Manager. Sélectionnez une toolbox qui vous intéresse et que vous souhaitez installer. Assurez-vous de consulter sa description pour comprendre ses fonctionnalités.

— **Installation de la Toolbox :** Une fois la toolbox choisie, suivez les instructions pour l'installer via l'ATOMS Manager. Cela implique généralement de sélectionner la toolbox dans la liste et de cliquer sur le bouton d'installation ou de suivre les étapes indiquées spécifiquement pour cette toolbox.

— **Vérification de l'Installation :** Après avoir installé la toolbox, vérifiez qu'elle est correctement installée en consultant la liste des toolboxes installées dans l'ATOMS Manager.

— **Exploration des Fonctionnalités :** Explorez les fonctionnalités de la toolbox nouvellement installée. Testez quelques-unes de ses fonctionnalités principales pour vous familiariser avec ses capacités et son utilisation.

Installation Hors Ligne

Instructions :

— **Téléchargement du Fichier de Toolbox :** Procurez-vous le fichier de la toolbox fourni, généralement au format ".zip" ou ".tzb". Assurez-vous de l'avoir enregistré sur votre ordinateur dans un emplacement facilement accessible.

— **Ouverture de l'ATOMS Manager :** Lancez Scilab et ouvrez l'ATOMS Manager. Pour ce faire, cliquez sur "Applications" dans la barre de menu de Scilab, puis sélectionnez "ATOMS". Cette action ouvrira l'interface de gestion des toolboxes.

— **Installation de la Toolbox :** Cherchez l'option "Installer une nouvelle toolbox (hors ligne)" dans l'ATOMS Manager. Choisissez cette option et parcourez votre système de fichiers pour sélectionner le fichier de la toolbox que vous avez téléchargé. Suivez les étapes pour installer la toolbox depuis votre ordinateur.

— **Vérification de l'Installation :** Après avoir terminé l'installation, vérifiez que la toolbox est correctement installée en consultant la liste des toolboxes installées dans l'ATOMS Manager.

— **Test des Fonctionnalités :** Explorez les fonctionnalités de la toolbox nouvellement installée. Testez quelques fonctionnalités clés pour vous familiariser avec son utilisation et ses capacités.

Installation via la Ligne de Commande :

Instructions :

— **Ouverture de la Console Scilab :** Lancez Scilab sur votre ordinateur pour accéder à la console.

— **Utilisation de la Fonction "atomsInstall()" :** Dans la console Scilab, saisissez la commande `atomsInstall('nom de la toolbox')`, en remplaçant "nom de la toolbox" par le nom spécifique de la toolbox que vous souhaitez installer. Appuyez sur Entrée pour exécuter la commande.

— **Vérification de l'Installation :** Après avoir exécuté la commande **atomsInstall()**, vérifiez dans la console que l'installation s'est déroulée correctement. Assurez-vous de ne pas avoir rencontré d'erreurs lors de l'installation.

— **Exploration du Contenu de la Toolbox :** Une fois installée, explorez le contenu de la toolbox nouvellement installée. Utilisez les fonctions ou les éléments disponibles dans la toolbox pour comprendre ses fonctionnalités et son utilisation.

Chapitre 2

Syntaxe et Éléments de Base

2.1 Variables et types de données

Dans ce premier chapitre, nous allons explorer les fondamentaux de Scilab en nous concentrant sur les variables et les types de données. Les variables sont des éléments cruciaux en programmation car elles nous permettent de stocker et de manipuler des informations. Scilab offre une grande souplesse en termes de types de données, ce qui nous permet de travailler avec différents types de variables, tels que les nombres, les chaînes de caractères et les tableaux.

2.1.1 Nombres

Scilab prend en charge différents types de nombres, tels que les entiers, les nombres réels et les nombres complexes. Les entiers sont des nombres sans partie décimale, tandis que les nombres réels peuvent avoir une partie décimale. Les nombres complexes, quant à eux, possèdent à la fois une partie réelle et une partie imaginaire.

Pour déclarer une variable contenant un nombre entier, vous pouvez utiliser la syntaxe suivante :

```
a = 8;
```

```
b = 3.14;
```

Pour déclarer une variable contenant un nombre complexe, vous pouvez utiliser la syntaxe suivante :

```
c = 6+ 5*%i;
```

2.1.2 Chaînes de caractères

Les chaînes de caractères servent à représenter du texte dans Scilab. Elles sont définies en utilisant des guillemets simples ou doubles. Voici quelques exemples :

```
nom = 'Fred Mk';
message = "Bonjour, comment ça va ?";
```

Il est également possible de concaténer des chaînes de caractères en utilisant l'opérateur de concaténation '+'. Par exemple :

```
prenom = 'Fred';
nom = 'Mk';
nom_complet = prenom + ' ' + nom;
```

Dans cet exemple, la variable 'nom_complet' contiendra la chaîne de caractères "Fred Mk".

2.1.3 Tableaux

Les tableaux sont des structures de données permettant de stocker plusieurs valeurs dans une seule variable. Scilab prend en charge les tableaux unidimensionnels (vecteurs) et multidimensionnels (matrices).

Pour déclarer un vecteur, vous pouvez utiliser la syntaxe suivante :

```
vecteur = [1, 2, 3, 4, 5];
```

Pour accéder aux éléments d'un vecteur, utilisez l'indexation. Par exemple, pour accéder au deuxième élément du vecteur :

```
deuxieme_element = vecteur(2);
```

Pour déclarer une matrice, utilisez la syntaxe suivante :

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

2.1.4 Accès aux éléments d'une matrice

Pour accéder aux éléments d'une matrice, utilisez l'indexation en spécifiant le numéro de ligne et de colonne :

```
element = matrice(2, 3);
```

2.1.5 Types de données spéciaux

En plus des types de données précédemment mentionnés, Scilab propose également des types de données spéciaux tels que les booléens, les listes et les structures. Les booléens représentent les valeurs de vérité (vrai ou faux). Les listes sont des collections ordonnées d'éléments, tandis que les structures sont des collections d'éléments associés à des noms.

Pour déclarer un booléen :

```
vrai = %T;
faux = %F;
```

Pour déclarer une liste :

```
liste = [1, 2, 3, 4, 5];
```

Pour accéder aux éléments d'une liste, utilisez l'indexation comme pour les vecteurs.

Pour déclarer une structure :

```
structure = struct('nom', 'Fred', 'age', 30, 'ville', 'Paris');
```

Pour accéder aux éléments d'une structure :

```
nom = structure.nom;
age = structure.age;
ville = structure.ville;
```

2.2 Opérations mathématiques

Les opérations mathématiques constituent le cœur de Scilab. Ce logiciel offre une large gamme de fonctions mathématiques permettant de réaliser des calculs complexes et de résoudre des problèmes mathématiques de manière efficace. Dans cette section, nous allons explorer différentes opérations mathématiques disponibles dans Scilab et apprendre à les utiliser.

2.2.1 Opérations arithmétiques

Scilab prend en charge les opérations arithmétiques de base telles que l'addition, la soustraction, la multiplication et la division. Vous pouvez utiliser les opérateurs +, -, * et / pour effectuer ces opérations. Par exemple, pour ajouter deux nombres :

```
a = 5;
b = 3;
c = a + b;
disp(c);
```

2.2.2 Fonctions mathématiques

Scilab propose un ensemble étendu de fonctions mathématiques de base. Voici quelques exemples :

- Fonctions trigonométriques :
  - sin(x) : Sinus de x (x en radians).
  - cos(x) : Cosinus de x (x en radians).
  - tan(x) : Tangente de x (x en radians).
  - asin(x) : Arc sinus de x.
  - acos(x) : Arc cosinus de x.
  - atan(x) : Arc tangente de x.
- Fonctions exponentielles et logarithmiques :
  - exp(x) : Exponentielle de x.
  - log(x) : Logarithme népérien de x.
  - log10(x) : Logarithme décimal de x.
- Fonctions racines :
  - sqrt(x) : Racine carrée de x.
- Fonctions trigonométriques hyperboliques :
  - sinh(x) : Sinus hyperbolique de x.
  - cosh(x) : Cosinus hyperbolique de x.
  - tanh(x) : Tangente hyperbolique de x.
- Fonctions arrondies :
  - ceil(x) : Plus petit entier supérieur ou égal à x.
  - floor(x) : Plus grand entier inférieur ou égal à x.
  - round(x) : Arrondi de x.
- Fonctions aléatoires :
  - rand() : Génère un nombre aléatoire entre 0 et 1.
  - rand(n, m) : Génère une matrice de dimensions n x m de nombres aléatoires entre 0 et 1.

2.2.3 Opérations matricielles

Scilab offre également des fonctionnalités avancées pour effectuer des opérations sur les matrices. Vous pouvez créer des matrices, les manipuler, effectuer des opérations matricielles telles que l'addition, la soustraction, la multiplication, etc.

Pour créer une matrice :

```
A = [1, 2, 3; 4, 5, 6; 7, 8, 9];
disp(A);
```

Pour ajouter deux matrices :

```
A = [1, 2; 3, 4];
B = [5, 6; 7, 8];
C = A + B;
disp(C);
```

Scilab propose également des fonctions pour des opérations matricielles avancées telles que le calcul de l'inverse d'une matrice, la décomposition en valeurs singulières, etc. Ces fonctions, disponibles dans le module d'algèbre linéaire, peuvent être utilisées pour résoudre des problèmes mathématiques complexes.

2.3 Écrire et exécuter un script

On peut écrire dans un fichier **fich1** une suite de commandes et les faire exécuter par l'instruction :

```
-->exec('fich1') // ou encore
exec("fich1")
```

On peut aussi charger le fichier par l'intermédiaire du menu "fichier", puis l'éxécuter par l'item "Exécuter".

Voici un exemple que l'on nommera "script.sce" :

```
// mon premier script Scilab
a = input("Rentrer la valeur de a : ");
b = input("Rentrer la valeur de b : ");
n = input("Nombre d'intervalles n : ");
// calcul des abscisses
x = linspace(a, b, n + 1);
```

```
// calcul des ordonnees
y = exp(-x) .* sin(4 * x);
// representation graphique
plot(x, y);
xtitle("y = exp(-x) * sin(4x)", "x", "y");
```

2.4 Entrées-sorties et fichiers

2.4.1 Lecture par le clavier et affichage

La saisie au clavier se réalise en utilisant la commande `input`

```
--> n=input("entrer la dimension n")
entrer la dimension n
--> 2
n =
2.
--> a=input("entrer une matrice")
entrer une matrice
--> [1, 2; 3, 4]
a =
1. 2.
3. 4.
```

L'entrée d'une chaîne de caractères se fait également avec la fonction `input`, en ajoutant un second paramètre qui est la constante `"string"`.

```
-->  chaine = input("Entrez une chaîne de caractères: ", "string")
Entrez une chaîne de caractères: "Bonjour"
-->  print(chaine)
"Bonjour"
```

Affichage de données avec `disp`

```
--> a=[1, 2; 3, 4]; b="coucou"; c=2-sqrt(3);
--> disp(a,b,c)
1. 2.
3. 4.
```

```
"coucou"
 0.2679492
```

On peut aussi utiliser l'instruction `printf` qui est en fait celle qui est héritée du langage C.

Affichage avec la fonction `printf`

```
--> a=50; b=cos(a);
--> printf('le cosinus de %f \n est %f', a, b)
le cosinus de 50.000000
 est 0.964966
```

2.4.2 Utilisation de fichiers

Pour lire à partir d'un fichier, on peut utiliser la fonction `read` en spécifiant le nom du fichier ainsi que d'autres paramètres

De même, pour écrire dans un fichier, on peut utiliser la fonction `write` en spécifiant le nom du fichier ainsi que les données à écrire

```
// Exemple de stockage de données utilisant une liste de listes
a = [1, 2, 3; 4, 5, 6];
b = ["Université"; "du Havre"];
// Écriture des données dans des fichiers
write("fic1.txt", a);
write("fic2.txt", b);
// Lecture des données depuis les fichiers
c = read("fic1.txt", 2, 3);
d = read("fic2.txt");
// Affichage des données lues
disp(c);
disp(d);
```

2.5 La programmation en Scilab

Scilab est un langage de programmation complet axé sur la manipulation de matrices et doté d'une bibliothèque graphique intégrée. Contrairement aux langages traditionnels, il n'exige pas de déclarations explicites, simplifiant ainsi

la création de programmes. Cependant, son principal inconvénient réside dans le fait qu'il n'est pas compilé directement en langage machine, ce qui peut entraîner des temps d'exécution plus longs par rapport aux langages compilés. Malgré cela, Scilab offre la possibilité d'interfacer avec des langages comme Fortran 77 et C, offrant ainsi aux développeurs une flexibilité supplémentaire. La structure typique d'un programme Scilab comprend une phase d'initialisation, la définition de fonctions personnalisées, le corps principal du programme contenant les calculs, et l'affichage des résultats. Cette organisation facilite la compréhension, la maintenance et la modification du code.

2.5.1 Branchements et boucles

2.5.1.1 If-instructions conditionnelles

```
1 if (%t) then
2     disp("Hello!")
3     end
```

Dans ce code :

— L'instruction `if` évalue une condition. Ici, la condition est `%t`, qui signifie "vrai" en Scilab.

— Si la condition est vraie, le bloc de code à l'intérieur de l'instruction `if` est exécuté.

— Dans ce cas, la fonction `disp` est utilisée pour afficher le message "Hello !" à l'écran.

— Enfin, l'instruction `end` marque la fin du bloc de code associé à l'instruction `if`.

Ainsi, lors de l'exécution de ce code en Scilab, le message "Hello !" sera affiché à l'écran car la condition `%t` est toujours vraie.

2.5.1.2 Test avec alternative

Nous utilisons l'instruction `if` en Scilab pour effectuer une action conditionnelle avec une alternative. comme suit :

```
1 if (%f) then
2     disp("Hello!")
3 else
4     disp("Bye!")
5 end
```

Dans ce code :

— L'instruction `if` évalue une condition. Ici, la condition est `%f`, qui signifie "faux" en Scilab.

— Si la condition est vraie, le bloc de code à l'intérieur de l'instruction `if` est exécuté.

— Sinon, le bloc de code à l'intérieur de l'instruction `else` est exécuté.

— Dans ce cas, la fonction `disp` est utilisée pour afficher le message "Bye!" à l'écran, car la condition `%f` est fausse.

— L'instruction `end` marque la fin du bloc de code associé à l'instruction `if`.

Ainsi, lors de l'exécution de ce code en Scilab, le message "Bye!" sera affiché à l'écran car la condition `%f` est fausse.

Dans Scilab, il est possible d'enchaîner plusieurs tests conditionnels en utilisant l'instruction `elseif`. L'expression de sélection entre parenthèses après `if` doit être une expression booléenne. Il est important de se rappeler que pour une comparaison, il faut utiliser le double symbole `==` et non `=` qui est utilisé pour l'affectation.

Voici un exemple illustrant l'utilisation de `elseif` :

```
a = 5;
if a == 3 then
    disp("a est égal à 3");
elseif a == 5 then
    disp("a est égal à 5");
else
    disp("a n'est ni égal à 3 ni à 5");
end
```

Dans cet exemple :

— La variable `a` est initialisée à 5.

— La première condition vérifie si `a` est égal à 3. Si c'est le cas, le message "a est égal à 3" est affiché.

— Sinon, la deuxième condition vérifie si `a` est égal à 5. Si c'est le cas, le message "a est égal à 5" est affiché.

— Si aucune des conditions précédentes n'est vraie, le bloc de code associé à `else` est exécuté, affichant le message "a n'est ni égal à 3 ni à 5".

Ainsi, l'utilisation de `elseif` permet de définir plusieurs conditions alternatives dans une structure conditionnelle en Scilab.

2.5.1.3 Select-alternatives

```
day = input('Entrez un numéro de jour (1-7): ');

switch day
    case 1
        disp('Lundi');
    case 2
        disp('Mardi');
    case 3
        disp('Mercredi');
    case 4
        disp('Jeudi');
    case 5
        disp('Vendredi');
    case 6
        disp('Samedi');
    case 7
        disp('Dimanche');
    otherwise
        disp('Numéro de jour invalide.');
end
```

Ce programme demande à l'utilisateur d'entrer un numéro de jour (entre 1 et 7) et affiche ensuite le nom du jour correspondant. Par exemple, si l'utilisateur entre 3, le programme affichera "Mercredi". Si l'utilisateur entre un nombre en dehors de cette plage (par exemple, 9), le programme affichera "Numéro de jour invalide." grâce au bloc `otherwise`.

2.5.1.4 For-boucle

En Scilab, la boucle `for` est utilisée pour itérer sur une séquence d'éléments ou pour exécuter un bloc de code un nombre spécifié de fois. Voici sa syntaxe générale :

```
for variable = début:pas:fin
    instructions
end
```

- `variable` : C'est la variable qui sera utilisée comme indice de la boucle. Elle prendra successivement les valeurs de `début` à `fin` avec un pas spécifié.
- `début`, `pas`, `fin` : Ce sont des expressions scalaires qui déterminent la plage sur laquelle la boucle s'exécutera. La boucle commencera à `début`,

incrémentera ou décrémentera de `pas` à chaque itération, et s'arrêtera lorsque la valeur atteint ou dépasse `fin`.

— `instructions` : Ce sont les instructions à exécuter à chaque itération de la boucle. Elles peuvent inclure n'importe quel code valide en Scilab.

Voici un exemple d'utilisation de la boucle `for` en Scilab :

```
// Afficher les carrés des nombres de 1 à 5
for i = 1:5
    disp(i^2);
end
```

Dans cet exemple, la boucle `for` itère sur les valeurs de `i` de 1 à 5. À chaque itération, la valeur de `i` est mise au carré et affichée à l'aide de la fonction `disp`.

C'est ainsi que fonctionne la boucle `for` en Scilab. Elle est utile pour itérer sur des séquences de valeurs et automatiser l'exécution répétée de blocs de code.

2.5.1.5 While-boucle

La boucle `while` en Scilab répète un bloc de code tant qu'une condition est vraie. Elle commence par initialiser une variable de contrôle, évalue la condition à chaque itération, et exécute le bloc de code tant que la condition reste vraie. Pour éviter les boucles infinies, il est essentiel que la condition devienne fausse à un moment donné.

```
// Initialisation de la variable de contrôle
compteur = 1;
// Début de la boucle while
while compteur <= 5
    disp("Compteur :", compteur);
    // Incrémentation du compteur
    compteur = compteur + 1;
end
// Sortie de la boucle
disp("Fin de la boucle");
```

Ce code utilise une boucle `while` pour afficher les nombres de 1 à 5 en incrémentant une variable de comptage, puis imprime `Fin de la boucle` une fois la boucle terminée.

2.5.1.6 Break et continue-sorties

L'instruction `break` permet de sortir d'une boucle lorsqu'une condition est satisfaite, tandis que "continue" signale qu'il faut passer à l'itération suivante

sans exécuter le reste du code dans la boucle actuelle. Ces instructions peuvent être évitées en faveur d'une analyse plus approfondie de l'algorithme pour une meilleure lisibilité et vérifiabilité du code.

```
s=0; i=0;
while (i<=10)
    if (modulo(i,2)==0) then
        i=i+1;
        continue;
    end
    s=s+i;
    i=i+1;
end
```

La construction précédente, qui est un exemple de construction algorithmique à ne pas suivre, s'obtient plus simplement et efficacement avec la seule instruction : `sum(1:2:10)`.

2.5.2 Les fonctions

La démarche fondamentale de développement en Scilab repose sur la décomposition fonctionnelle, impliquant la construction progressive de programmes par l'élaboration de briques de base fonctionnelles. Ces briques sont ensuite assemblées ou utilisées pour créer des fonctions plus complexes et abstraites.

Une fonction en Scilab est utilisée comme suit :

```
outvar = myfunction(invar)
```

où :

- `myfunction` est le nom de la fonction ;
- `invar` représente les arguments d'entrée nécessaires à l'exécution de la fonction ;
- `outvar` est le résultat calculé par la fonction.

Scilab propose déjà de nombreuses fonctions prédéfinies, telles que les fonctions mathématiques classiques (`sin`, `cos`, etc.).

Les fonctions Scilab peuvent accepter un nombre arbitraire d'arguments d'entrée et de sortie, comme illustré ci-dessous :

```
[o1, o2, ..., on] = myfunction(i1, i2, ..., in)
```

Supposons que nous voulions écrire une fonction pour calculer la somme des éléments d'une liste en Scilab. Nous pouvons créer une fonction nommée `sommeListe` comme suit :

```
function somme = sommeListe(liste)
    somme = sum(liste);
endfunction
```

Ensuite, nous pouvons l'utiliser pour calculer la somme des éléments d'une liste donnée :

```
liste = [1, 2, 3, 4, 5];
resultat = sommeListe(liste);
disp(resultat); // Affiche 15

```

2.5.2.1 Définir une fonction

La définition d'une fonction en Scilab peut être laborieuse en fonction de sa complexité. Plusieurs méthodes sont disponibles pour saisir la fonction : la saisie directe dans l'environnement Scilab en utilisant le mot-clé `endfunction` pour l'évaluer, l'utilisation de l'éditeur intégré avec l'option `Exécuter/Charger dans Scilab` pour charger la fonction, ou la sauvegarde du code dans un fichier `.sci` ou `.sce` suivi de l'utilisation de la fonction `exec` pour l'importer dans l'environnement interactif.

```
// Définition d'une fonction qui calcule le carré d'un nombre
function resultat = carre(nombre)
resultat = nombre^2;
endfunction\\
```

Cette fonction nommée `carre` prend un argument `nombre` en entrée et retourne le carré de ce nombre.

```
// Utilisation de la fonction "carre"
x = 5;
resultat = carre(x);
disp(resultat); // Affiche 25
```

Nous utilisons la fonction `carre` en lui passant la valeur `5` comme argument. Le résultat est stocké dans la variable `resultat` et affiché.

```
// Définition d'une fonction qui calcule la somme de deux nombres
function somme = addition(nombre1, nombre2)
    somme = nombre1 + nombre2;
endfunction
```

Cette fonction nommée `addition` prend deux arguments `nombre1` et `nombre2` en entrée et retourne leur somme.

```
// Utilisation de la fonction "addition"
a = 3;
b = 7;
resultat = addition(a, b);
disp(resultat); // Affiche 10
```

Nous utilisons la fonction `addition` en lui passant deux nombres comme arguments. Le résultat est stocké dans la variable `resultat` et affiché.

2.5.2.2 Construction dynamique de fonctions en Scilab

Scilab propose plusieurs méthodes pour construire des fonctions de manière dynamique, notamment avec les commandes `deff`, `evstr` et `execstr`.

— **Utilisation de la commande `deff` :**
La commande `deff` permet de définir des fonctions "in-line" en une seule instruction en prenant des chaînes de caractères comme argument d'entrée où la fonction sera décrite.

```
deff('y = cube(x)', 'y = x^3' );
```

Cette instruction utilise la commande `deff` pour définir une fonction "in-line" nommée `cube` qui prend un argument x en entrée et retourne son cube. La chaîne de caractères $y = x^3$ décrit la fonction à définir.

— **Utilisation de `evstr` pour définir dynamiquement une fonction**

```
// Définition de la chaîne de caractères contenant la
↪  fonction
expression = 'function resultat = carre(nombre), resultat =
↪  nombre^2; endfunction';

// Évaluation et exécution de la chaîne de caractères avec
↪  evstr
```

```
evstr(expression);

// Appel de la fonction avec un exemple
nombre = 5;
res = carre(nombre);
disp('Le carré de ', nombre, ' est : ', res);

```

Cette séquence de commandes utilise `evstr` pour évaluer la chaîne de caractères `expression`, qui contient la définition d'une fonction pour calculer le carré d'un nombre.

— **Utilisation de la commande `execstr` :**
La commande `execstr` permet d'exécuter une chaîne de caractères en tant qu'instruction Scilab.

```
// Définition de la chaîne de caractères contenant la
↪  fonction
expression = 'function resultat = carre(nombre), resultat =
↪  nombre^2; endfunction';

// Évaluation et exécution de la chaîne de caractères
execstr(expression);

// Appel de la fonction avec un exemple
nombre = 5;
res = carre(nombre);
disp('Le carré de ', nombre, ' est : ', res);

```

Cette séquence de commandes utilise `execstr` pour exécuter la chaîne de caractères `instruction`, qui contient la définition d'une fonction pour additionner deux nombres.

Travaux Pratique 2 : Les bases de Scilab

Min et Max

Scilab sait trouver le plus grand élément (c'est la fonction max) et le plus petit élément (c'est la fonction min) d'un vecteur ou d'une matrice.

- Écrire une commande permettant de déterminer le plus grand multiple de 7 inférieur ou égal à 100.
- Écrire une commande permettant de déterminer le plus petit multiple de 7 supérieur ou égal à 1000.
- Écrire une commande permettant d'entrer trois entiers naturels n, p et q, puis de déterminer le plus petit entre le plus grand multiple de p inférieur ou égal à n et le plus grand multiple de q inférieur ou égal à n.

Autour de sum et cumsum

On suppose que n a été créé numériquement.

- Écrire une ligne de commandes qui renvoie la somme des n premiers entiers naturels non nuls.
- Écrire une ligne de commandes renvoyant la somme $\sum_{k=1}^{n} \frac{1}{k}$.
- Que calculent les commandes suivantes ?
  - $x = \text{ones}(1, n); y = \text{cumsum}(x)$
  - $x = \text{ones}(1, n); y = \text{sum}(\text{cumsum}(x))$
  - $\text{x} = \text{ones}(1, \text{n}); \text{y} = \text{sum}(\text{cumsum}(\text{cumsum}(\text{x})))$

La boucle for

On vient de voir que l'on pouvait écrire $\sum_{k=1}^{n} \frac{1}{k}$ avec la commande sum. Sans utiliser la commande sum, proposer une suite de commandes utilisant une boucle for et permettant de calculer, pour un entier naturel donné la valeur de

$$u_n = \sum_{k=1}^{n} \frac{1}{k} - \ln(n)$$

pour $n = 100, n = 1000, n = 10000$.

La boucle while

Écrire une ligne de commandes permettant de déterminer le plus petit entier naturel n pour lequel

$$\sum_{k=1}^{n} \frac{1}{k} > 10$$

Suite récurrente

On définit la suite (u_n) par $u_0 \in \mathbb{R}$ et pour $n \in \mathbb{N}$

$$u_{n+1} = \frac{1}{2}\sqrt{3 + u_n^2}.$$

— Écrire une ligne de commandes demandant une valeur de n et u_0 et permettant de calculer le terme u_n.

— Ajouter une commande permettant de calculer la somme $\sum_{k=0}^{n} u_k$.

— On admet que la suite (u_n) converge et que sa limite vaut 1. Écrire une suite de commandes qui permette de déterminer le plus petit entier naturel n pour lequel $|u_n - 1| \leq 10^{-3}$, u_0 étant entré par l'utilisateur.

Correction de TP 2 : Les bases de Scilab

Min et Max

— Le plus grand multiple de 7 inférieur ou égal à 1000 :

```
x= 7 * floor(1000 / 7);
disp(x);
```

— Le plus petit multiple de 7 supérieur ou égal à 1000 :

```
x = ceil(1000 / 7) * 7;
disp("Le plus petit multiple de 7 supérieur ou égal à 1000 est: "
↪  + string(x));
```

— Fonction pour déterminer le plus petit entre le plus grand multiple de p inférieur ou égal à n et le plus grand multiple de q inférieur ou égal à n :

```
function [resultat] = trouverPlusPetitMultiple(n, p, q)
    plusGrandMultipleP = floor(n / p) * p;
    plusGrandMultipleQ = floor(n / q) * q;
    if plusGrandMultipleP <= plusGrandMultipleQ then
        resultat = plusGrandMultipleP;
    else
        resultat = plusGrandMultipleQ;
    end
endfunction

n = input("Entrez la valeur de n: ");
p = input("Entrez la valeur de p: ");
q = input("Entrez la valeur de q: ");
resultat = trouverPlusPetitMultiple(n, p, q);
disp("Le plus petit entre le plus grand multiple de p et q
↪  inferieurs ou egaux a n est: ");
disp(resultat);
```

Autour de sum et cumsum

— Somme des n premiers entiers naturels non nuls :

```
sum_n_premiers_entiers = sum(1:n);
```

— Somme $\sum_{k=1}^{n} \frac{1}{k}$:

```
somme_inverse = sum(1./1:n);
```

— Explications pour les commandes :

- `x=ones(1, n) ; y=cumsum(x)` : Crée un vecteur de n éléments, tous égaux à 1, et renvoie le vecteur cumulatif.
- `x=ones(1, n) ; y=sum(cumsum(x))` : Comme précédemment, mais renvoie la somme de tous les éléments du vecteur cumulatif.
- `x=ones(1, n) ; y=sum(cumsum(cumsum(x)))` : Crée un vecteur de n éléments égaux à 1, applique la fonction `cumsum` deux fois, puis renvoie la somme de tous les éléments du deuxième vecteur cumulatif.

La boucle for

Calcul de $u_n = \sum_{k=1}^{n} \frac{1}{k} - \ln(n)$ pour $n = 100, 1000, 10000$:

```
n_values = [100, 1000, 10000];

for n = n_values
    somme_inverse = 0;
    for k = 1:n
        somme_inverse = somme_inverse + 1/k;
    end
    u_n = somme_inverse - log(n);
    disp(u_n);
end
```

La boucle while

Calcul du plus petit entier naturel n tel que $\sum_{k=1}^{n} \frac{1}{k} > 10$:

```
n = 1;
somme_inverse = 0;

while somme_inverse <= 10
    somme_inverse = somme_inverse + 1 / n;
    n = n + 1;
end

disp(n - 1); % Affiche le plus petit entier n
```

Suite récurrente

Calcul de u_n et $\sum_{k=0}^{n} u_k$, puis détermination du plus petit entier naturel n tel que $|u_n - 1| \leq 10^{-3}$:

```
function u_n = terme_un(n, u_0)
    u_n = u_0;
    for i = 1:n
        u_n = 0.5 * sqrt(3 + u_n^2);
    end
endfunction

function somme_u_n = somme_u_n(n, u_0)
    somme_u_n = 0;
    for i = 0:n
        somme_u_n = somme_u_n + terme_un(i, u_0);
    end
endfunction

function smallest_n(u_0)
    n = 0;
    while abs(terme_un(n, u_0) - 1) > 1e-3 do
        n = n + 1;
    end
    disp(n);
endfunction
```

```
22
23 u_0 = input("Entrez la valeur initiale u_0: ");
24 n = input("Entrez la valeur de n: ");
25 u_n = terme_un(n, u_0);
26 disp(u_n);
27 sum_u_n = somme_u_n(n, u_0);
28 disp(sum_u_n);
29 smallest_n(u_0);
```

Chapitre 3

Manipulation de données

3.1 Tableaux et matrices

Les tableaux et les matrices sont des structures de données essentielles dans Scilab. Ils permettent de stocker et de manipuler des ensembles de valeurs de manière efficace. Dans cette section, nous allons explorer les différentes fonctionnalités offertes par Scilab pour travailler avec des tableaux et des matrices.

3.2 Opérations sur les matrices

Les matrices sont des structures de données essentielles en Scilab. Elles permettent de stocker et de manipuler des ensembles de valeurs numériques de manière efficace. Dans cette section, nous allons explorer les différentes opérations que nous pouvons effectuer sur les matrices en utilisant Scilab.

3.2.1 Création de matrices

Avant de pouvoir effectuer des opérations sur les matrices, nous devons d'abord les créer. Scilab offre plusieurs méthodes pour créer des matrices.

La méthode la plus simple consiste à utiliser la fonction zeros pour créer une matrice remplie de zéros. Par exemple, pour créer une matrice de taille 3x3 remplie de zéros, nous pouvons utiliser la commande suivante :

```
A = zeros(3, 3);
```

Nous pouvons également utiliser la fonction ones pour créer une matrice remplie de uns, ou la fonction rand pour créer une matrice remplie de nombres aléatoires entre 0 et 1.

Il est également possible de créer des matrices en spécifiant directement leurs valeurs. Par exemple, pour créer une matrice 2x2 avec les valeurs [1, 2 ; 3, 4], nous pouvons utiliser la commande suivante :

```
B = [1, 2; 3, 4];
```

3.2.2 Opérations arithmétiques sur les matrices

Scilab permet d'effectuer des opérations arithmétiques sur les matrices de manière simple et efficace. Les opérations arithmétiques de base, telles que l'addition, la soustraction, la multiplication et la division, peuvent être effectuées directement sur les matrices.

Par exemple, pour ajouter deux matrices A et B, nous pouvons utiliser l'opérateur + :

```
C = A + B;
```

De même, pour soustraire deux matrices A et B, nous pouvons utiliser l'opérateur - :

```
D = A - B;
```

Pour multiplier deux matrices A et B, nous pouvons utiliser l'opérateur * :

```
E = A * B;
```

Il est important de noter que la multiplication de matrices en Scilab suit les règles de l'algèbre linéaire. Cela signifie que le nombre de colonnes de la première matrice doit être égal au nombre de lignes de la deuxième matrice.

3.2.3 Opérations sur les éléments des matrices

En plus des opérations arithmétiques, Scilab permet également d'effectuer des opérations sur les éléments individuels des matrices. Par exemple, nous pouvons calculer la somme des éléments d'une matrice en utilisant la fonction sum :

```
total = sum(A);
```

Nous pouvons également calculer la moyenne des éléments d'une matrice en utilisant la fonction mean :

```
average = mean(A);
```

3.2.4 Transposition de matrices

La transposition d'une matrice consiste à échanger ses lignes et ses colonnes. Scilab offre une fonction transpose pour effectuer cette opération. Par exemple, pour transposer une matrice A, nous pouvons utiliser la commande suivante :

```
A_transpose=A';
```

La transposition d'une matrice peut être utile dans de nombreux cas, notamment pour effectuer des opérations telles que la multiplication de matrices ou la résolution de systèmes linéaires.

3.2.5 Inversion de matrices

L'inversion d'une matrice consiste à trouver une autre matrice qui, lorsqu'elle est multipliée par la matrice d'origine, donne une matrice identité. Scilab offre une fonction inv pour inverser une matrice. Par exemple, pour inverser une matrice A, nous pouvons utiliser la commande suivante :

```
A = [1, 2; 3, 4];
A_inverse = inv(A);
```

Il est important de noter que toutes les matrices ne sont pas inversibles. Une matrice est inversible si et seulement si son déterminant est différent de zéro.

3.2.6 Concaténation de matrices

Scilab permet également de concaténer des matrices, c'est-à-dire de les joindre pour former une matrice plus grande. La concaténation peut être effectuée horizontalement ou verticalement.

Pour concaténer deux matrices horizontalement, vous pouvez utiliser des crochets carrés pour effectuer la concaténation :

```
C = [A, B];
```

Pour réaliser la concaténation verticale de deux matrices, il suffit d'employer la syntaxe suivante :

```
C = [A; B];
```

La concaténation de matrices peut être utile dans de nombreux cas, notamment pour combiner des ensembles de données ou pour créer des matrices plus grandes à partir de matrices plus petites.

3.3 Manipulation de vecteurs

Les vecteurs sont des structures de données essentielles en mathématiques et en programmation, et ils jouent un rôle important dans de nombreuses applications scientifiques et techniques. Scilab offre de puissantes fonctionnalités pour créer, manipuler et analyser des vecteurs, ce qui en fait un outil précieux pour les scientifiques, les ingénieurs et les chercheurs.

3.3.1 Création de vecteurs

La création de vecteurs est une opération courante dans Scilab. Il existe plusieurs façons de créer des vecteurs, en fonction de vos besoins spécifiques. Voici quelques-unes des méthodes les plus couramment utilisées :

3.3.1.1 Vecteurs numériques

Pour créer un vecteur numérique, vous pouvez utiliser la fonction linspace qui génère un vecteur de valeurs uniformément espacées entre une valeur de départ et une valeur de fin. Par exemple, pour créer un vecteur de 10 valeurs allant de 0 à 1, vous pouvez utiliser la commande suivante :

```
x = linspace(0, 1, 10);
```

Vous pouvez également créer un vecteur en spécifiant manuellement les valeurs à l'aide de la notation entre crochets. Par exemple, pour créer un vecteur contenant les valeurs 1, 2, 3, vous pouvez utiliser la commande suivante :

```
x = [1, 2, 3];
```

3.3.1.2 Vecteurs aléatoires

Scilab offre également des fonctions pour générer des vecteurs aléatoires. Par exemple, la fonction rand génère un vecteur de nombres aléatoires compris entre 0 et 1. Vous pouvez spécifier la taille du vecteur en utilisant les arguments de la fonction. Voici un exemple :

```
x = rand(1, 10);
```

Cette commande génère un vecteur de 10 valeurs aléatoires entre 0 et 1.

3.3.2 Accès aux éléments d'un vecteur

Une fois que vous avez créé un vecteur, vous pouvez accéder à ses éléments individuels en utilisant leur indice. Les indices commencent à 1 pour le premier élément et augmentent de 1 pour chaque élément suivant. Par exemple, pour accéder au troisième élément d'un vecteur x, vous pouvez utiliser la commande suivante :

```
x(3);
```

Cette commande renverra la valeur du troisième élément du vecteur x.

3.3.3 Opérations sur les vecteurs

Scilab offre de nombreuses opérations pour manipuler les vecteurs. Vous pouvez effectuer des opérations mathématiques telles que l'addition, la soustraction, la multiplication et la division sur les vecteurs. Par exemple, pour ajouter deux vecteurs x et y, vous pouvez utiliser la commande suivante :

```
z = x + y;
```

Cette commande créera un nouveau vecteur z contenant la somme des éléments correspondants des vecteurs x et y.

Vous pouvez également effectuer des opérations statistiques sur les vecteurs, telles que le calcul de la moyenne, de la variance et de l'écart-type. Scilab offre des fonctions spécifiques pour effectuer ces calculs. Par exemple, pour calculer la moyenne d'un vecteur x, vous pouvez utiliser la commande suivante :

```
mean_x = mean(x);
```

Cette commande renverra la moyenne des éléments du vecteur x.

3.3.4 Manipulation avancée des vecteurs

En plus des opérations de base, Scilab offre des fonctionnalités avancées pour manipuler les vecteurs. Vous pouvez effectuer des opérations de tri, de recherche et de filtrage sur les vecteurs.

La fonction gsort permet de trier les éléments d'un vecteur par ordre croissant ou décroissant. Par exemple, pour trier un vecteur x par ordre croissant, vous pouvez utiliser la commande suivante :

```
// La fonction gsort permet de trier les éléments d'un vecteur par
↪  ordre croissant ou décroissant. Par exemple, pour trier un vecteur
↪  x par ordre croissant, vous pouvez utiliser la commande suivante:

// Création d'un vecteur x

x = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5];

// Tri du vecteur x par ordre croissant

sorted_x_ascend = gsort(x, "g", "i");

// Affichage du vecteur trié par ordre croissant

disp("Vecteur trié par ordre croissant: ", sorted_x_ascend);

// Déclaration de la variable sorted_x_descend

sorted_x_descend = [];

// Tri du vecteur x par ordre décroissant

for i = 1:length(x)
    sorted_x_descend = [sorted_x_descend, max(x)];
    x = x(x <> max(x));
end

// Affichage du vecteur trié par ordre décroissant

disp("Vecteur trié par ordre décroissant:", sorted_x_descend);
```

Cette commande créera un nouveau vecteur **sorted_x** contenant les éléments de 'x' triés par ordre croissant.

La fonction **find** permet de rechercher les indices des éléments d'un vecteur qui satisfont une condition donnée. Par exemple, pour trouver les indices des éléments de X qui sont supérieurs à 5, vous pouvez utiliser la commande suivante :

```
x = [3, 7, 1, 8, 4, 5, 9, 2, 6];
indices_sup_5 = find(x > 5);
disp("Indices des éléments de x supérieurs à 5: ",
indices_sup_5);
```

Cette commande renverra un vecteur indices contenant les indices des éléments de x qui satisfont la condition.

La fonction filter permet de filtrer un vecteur en fonction d'une condition donnée. Par exemple, pour filtrer les éléments de x qui sont supérieurs à 5, vous pouvez utiliser la commande suivante :

```
filtered_x = x(x > 5);
```

Cette commande créera un nouveau vecteur filtered x contenant les éléments de x qui satisfont la condition.

Travaux Pratique 3 : Manipulation de données

La création de vecteurs et de matrices

On suppose que n a été créé numériquement (avec $n \geq 2$).

— Quelle est la différence entre les commandes suivantes ?

$$\mathrm{u} = \mathrm{n} : 3 : 1 \text{ et } \mathrm{linspace}(\mathrm{n}, 1, 3)$$

— Quel est le vecteur renvoyé par la commande suivante ?

$$\mathrm{u} = \mathrm{n} : -\mathrm{n}$$

— Quelle est la matrice renvoyée par la suite d'instructions qui suit ?

$$u = 1 : 3; v = ones(1, 3); \quad w = -1 : 2 : 4; x = [u; v; w]$$

La création de vecteurs

— Écrire une seule commande permettant de créer le vecteur $x = \left(5, \frac{5}{2}, \frac{5}{3}, \frac{5}{4}, \ldots, \frac{5}{10}\right)$ sans saisir un à un les éléments.

— Même question pour le vecteur $y = \left(1, \frac{1}{4}, \frac{1}{9}, \frac{1}{16}, \frac{1}{25}, \ldots, \frac{1}{100}\right)$.

— Même question pour le vecteur $z = \left(1, 2, 4, \ldots, 2^{10}\right)$.

La création de matrices

Écrire une ligne de commandes permettant de créer la matrice M dont les éléments diagonaux sont égaux à a et les autres éléments égaux à b, pour a, b et n entrés par l'utilisateur.

Matrices aléatoires

Génère deux matrices aléatoires de taille 3x3 et effectue l'addition, la multiplication, et la soustraction entre elles.

Matrices diagonales

Crée une matrice diagonale de taille 4x4 avec des valeurs de ton choix. Calcule ensuite son inverse.

Matrices singulières

Crée une matrice singulière (non inversible) de taille 2x2 et tente de trouver son inverse en Scilab.

Matrices spéciales

Explore la création de matrices spéciales comme la matrice nulle, la matrice identité, et la matrice unité en Scilab.

Résolution de système d'équations

Génère une matrice coefficient et un vecteur constant, puis utilise Scilab pour résoudre le système d'équations linéaires correspondant.

Manipulation de matrices

Utilise les fonctions Scilab pour extraire une sous-matrice à partir d'une matrice donnée

Répétition de matrices

Crée une matrice de taille 3x3 et utilise Scilab pour répéter ses éléments afin d'obtenir une matrice de taille 6x6.

Normes de matrices

Calcule la norme Frobenius et la norme spectrale d'une matrice de ton choix en utilisant les fonctions appropriées en Scilab

Correction de TP 3 : Manipulation de données

La création de vecteurs et de matrices

En Scilab, différentes commandes génèrent des séquences de nombres de façons différentes :

— `u = n:3:1` crée une séquence de nombres commençant à n, se terminant à 1 avec un pas de 3. Cependant, si $n > 1$, la séquence ne peut être formée correctement et donc `u` serait un vecteur vide.

— `linspace(n, 1, 3)` génère une séquence de 3 nombres linéairement répartis entre n et 1. Par exemple, si $n = 10$, cela produira la séquence 10 5.5 1, divisant linéairement l'intervalle de n à 1 en 3 parties égales.

— `u = n:-n` avec $n \geq 2$ tente de créer une séquence décroissante à partir de n. Cependant, cette formulation est incorrecte et produit un vecteur vide `[]` car elle tente de construire une séquence décroissante avec un pas de $-n$, ce qui ne génère aucune valeur valide.

Pour obtenir une séquence décroissante correcte en Scilab, utilisez `u = n:-1:1`, où n représente la valeur initiale. Par exemple, si $n = 6$, cela générera la séquence $6, 5, 4, 3, 2, 1$.

— Ensuite, la création d'une matrice `x` à partir de vecteurs se fait comme suit :

  — `u = 1:3` crée un vecteur `u` de 1 à 3.

  — `v = ones(1, 3)` crée un vecteur `v` contenant trois 1.

  — `w = -1:2:4` crée un vecteur `w` avec les valeurs -1, 1, et 3 dans ce cas, avec un pas de 2 jusqu'à 4.

  — Enfin, `x = [u; v; w]` crée une matrice `x` en concaténant verticalement les vecteurs `u`, `v` et `w`.

Donc, si nous résumons le contenu de chaque vecteur :

— `u` est $[1, 2, 3]$

— `v` est $[1, 1, 1]$

— `w` est $[-1, 1, 3]$

$$x = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 1 & 1 \\ -1 & 1 & 3 \end{pmatrix}$$

La Création de vecteurs

— En Scilab, pour obtenir le vecteur $x = (5, \frac{5}{2}, \frac{5}{3}, \frac{5}{4}, ..., \frac{5}{10})$, nous pouvons utiliser la fonction 1 :10 pour créer une séquence de nombres de 1 à 10 et ensuite inverser ces valeurs en les divisant par 10
x = 5 ./ (1 :10) ;

— En Scilab, pour obtenir le vecteur $y = (1, \frac{1}{4}, \frac{1}{9}, \frac{1}{16}, \frac{1}{25}..., \frac{1}{100})$, nous pouvons utiliser la fonction $y = \frac{1}{(1:10)^2}$ Chaque élément du vecteur est calculé en prenant l'inverse du carré de chaque nombre de 1 à 10.

— En Scilab, pour obtenir le vecteur $z = (1, 2, 4, ..., 2^{10})$, nous pouvons utiliser la fonction z = 2 . ^ (0 :10) ;

La Création de matrices

En utilisant Scilab, voici comment créer une matrice M avec des éléments diagonaux égaux à a a et les autres éléments égaux à b, en demandant à l'utilisateur d'entrer les valeurs de a, b et n :

```
function M = createMatrix(a, b, n)
    M = a * eye(n) + b * (ones(n, n) - eye(n));
endfunction

// Demander à l'utilisateur les valeurs de a, b et n

a = input("Entrez la valeur de a: ");
b = input("Entrez la valeur de b: ");
n = input("Entrez la valeur de n: ");

// Générer la matrice M avec les valeurs entrées par l'utilisateur

resultMatrix = createMatrix(a, b, n);
disp(resultMatrix);
```

Matrices aléatoire

Pour générer deux matrices aléatoires de taille 3x3 et effectuer différentes opérations matricielles, notamment l'addition, la multiplication et la soustraction entre ces deux matrices, on peut utiliser le programme suivant :

```
// Génération de deux matrices aléatoires 3x3
A = rand(3, 3);
B = rand(3, 3);

disp(A, 'Matrice A:');
disp(B, 'Matrice B:');

// Addition de matrices
C_addition = A + B;
disp(C_addition, 'Addition de A et B:');

// Multiplication de matrices
C_multiplication = A * B;
disp(C_multiplication, 'Multiplication de A et B:');

// Soustraction de matrices
C_soustraction = A - B;
disp(C_soustraction, 'Soustraction de A et B:');
```

Matrices diagonales

Pour créer une matrice diagonale de taille 4x4 avec les valeurs [2, 4, 6, 8] sur la diagonale, et calculer l'inverse de cette matrice diagonale en utilisant Scilab, le code suivant peut être employé :

```
// Création d'une matrice diagonale
D = diag([2, 4, 6, 8]);

disp(D, 'Matrice diagonale D:');

// Calcul de l'inverse de la matrice diagonale
D_inverse = inv(D);

disp(D_inverse, 'Inverse de la matrice diagonale D:');
```

Matrices singulières

Une matrice singulière S de taille 2x2 est générée avec des valeurs spécifiées. Par la suite, une tentative est faite pour calculer l'inverse de cette matrice en utilisant la fonction `inv`. Néanmoins, en raison de la singularité de la matrice, cette opération est vouée à l'échec, et le programme affiche un message indiquant l'impossibilité d'inverser la matrice.

```
// Création d'une matrice singulière
S = [1, 2; 2, 4];

% Affichage de la matrice singulière
disp(S, 'Matrice singulière S:');

try
  // Tentative de calcul de l'inverse de la matrice singulière
  S_inverse = inv(S);
  disp(S_inverse, 'Inverse de la matrice singulière S:');
catch
    disp('La matrice singulière S n\'est pas inversible.');
end
```

Matrices spéciales

Explorons la création de matrices spéciales comme la matrice nulle, la matrice identité et la matrice unité en Scilab. Voici le code correspondant

```
// Matrice nulle
matrice_nulle = zeros(3, 3);
disp(matrice_nulle, 'Matrice nulle:');

// Matrice identité
matrice_identite = eye(3);
disp(matrice_identite, 'Matrice identité:');

// Matrice unité
matrice_unite = ones(3, 3);
disp(matrice_unite, 'Matrice unité:');
```

Résolution de système d'équations

Pour générer une matrice de coefficients et un vecteur constant, puis résoudre le système d'équations linéaires correspondant, on utilise le programme suivant en Scilab :

```
// Génération de la matrice de coefficients
A = [2, 3; 4, -2];

// Génération du vecteur constant
B = [8; 6];

disp(A, 'Matrice de coefficients A:');
disp(B, 'Vecteur constant B:');

// Résolution du système d'équations linéaires
X = A \ B;

disp(X, 'Solution du système d''équations:');
```

Manipulation de matrices

Pour extraire une sous-matrice en utilisant l'opérateur de sélection " :" en Scilab, voici comment vous pourriez le faire :

```
// Crée une matrice de exemple
matrice_originale = [1, 2, 3; 4, 5, 6; 7, 8, 9];

// Spécifie les indices des lignes et colonnes pour extraire
la sous-matrice indices_lignes = 1:2; lignes 1 à 2 indices_colonnes =
↪  2:3; colonnes 2 à 3

// Utilise l'opérateur de sélection pour extraire la sous-matrice
sous_matrice = matrice_originale(indices_lignes, indices_colonnes);

// Affiche la sous-matrice résultante
disp(sous_matrice);
```

Répétition de matrices

Il est possible de créer une matrice de taille 3x3 en Scilab et de répéter ses éléments afin d'obtenir une matrice de taille 6x6 en utilisant la fonction "repmat". En spécifiant les arguments "2, 2", cette fonction répète la matrice tant sur les lignes que sur les colonnes, résultant en une matrice agrandie de dimensions 6x6.

```
// Crée une matrice de taille 3x3
matrice_3x3 = [1, 2, 3; 4, 5, 6; 7, 8, 9];

// Répète les éléments pour obtenir une matrice de taille 6x6
matrice_6x6 = repmat(matrice_3x3, 2, 2);

// Affiche la matrice résultante
disp(matrice_6x6);
```

Normes de matrices

Il est possible de calculer la norme Frobenius et la norme spectrale d'une matrice en utilisant la fonction 'norm'. Pour obtenir la norme Frobenius, utilisez l'argument "fro", et pour la norme spectrale, utilisez l'argument 2. Les résultats peuvent ensuite être affichés. Assurez-vous de remplacer la matrice par celle de votre choix.

```
// Crée une matrice de ton choix
matrice = [1, 2, 3; 4, 5, 6; 7, 8, 9];

// Calcule la norme Frobenius
norme_frobenius = norm(matrice, "fro");

// Calcule la norme spectrale
norme_spectrale = norm(matrice, 2);

// Affiche les résultats
disp("Norme Frobenius: " + string(norme_frobenius));
disp("Norme spectrale: " + string(norme_spectrale));
```

Chapitre 4

Graphiques et visualisation

4.1 Création de graphiques

La création de graphiques est l'une des fonctionnalités les plus puissantes de Scilab. Elle permet de représenter visuellement les données et les résultats des calculs mathématiques de manière claire et compréhensible. Dans cette section, nous allons explorer les différentes méthodes pour créer des graphiques dans Scilab.

4.1.1 Les types de graphiques

Scilab offre une grande variété de types de graphiques pour répondre à tous les besoins. Voici quelques-uns des types de graphiques les plus couramment utilisés :

4.1.1.1 Graphiques 2D

Les graphiques 2D sont l'un des types de visualisation les plus couramment utilisés. Ils permettent de représenter des données à deux dimensions, telles que des courbes, des diagrammes en barres, des histogrammes, etc. Scilab propose une variété de fonctions pour créer des graphiques 2D.

La fonction `plot` est l'une des fonctions les plus utilisées pour tracer des courbes. Elle permet de représenter graphiquement une série de points en les reliant par des segments de droite. Par exemple, pour tracer une courbe représentant la fonction $\cos(x)$ sur l'intervalle $[0, \pi]$, vous pouvez utiliser la commande suivante :

```
x = linspace(0, %pi, 100);
y = cos(x);
plot(x, y);
```

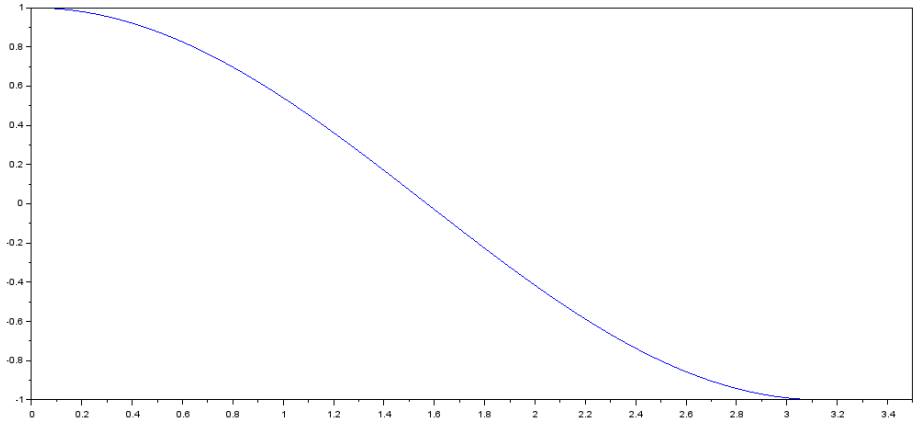

FIGURE 4.1 – Courbe représentant la fonction $\cos(x)$.

La fonction `bar` permet de créer des diagrammes en barres pour représenter des données discrètes. Par exemple, pour représenter le nombre de ventes mensuelles d'un produit, vous pouvez utiliser la commande suivante :

```
x = 1:5; // Données pour l'axe des abscisses
y = [10, 25, 15, 30, 20]; // Données pour l'axe des ordonnées
bar(x, y); // Tracer un diagramme en barres
```

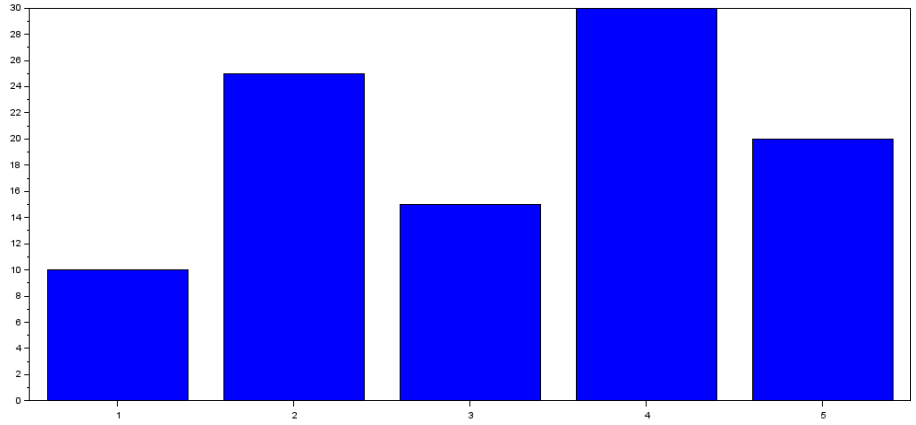

FIGURE 4.2 – Diagrammes en barres pour représenter des données discrètes.

4.1.1.2 Graphiques 3D

Les graphiques 3D permettent de représenter des données à trois dimensions. Ils sont particulièrement utiles pour visualiser des données volumineuses ou des fonctions à plusieurs variables. Scilab propose plusieurs fonctions pour créer des graphiques 3D.

La fonction `plot3d` permet de tracer des courbes en 3D. Par exemple, pour représenter une spirale en 3D, vous pouvez utiliser la commande suivante :

```
t = linspace(0, 10*%pi, 100);
x = cos(t);
y = sin(t);
z = t;
plot3d(x, y, z);
```

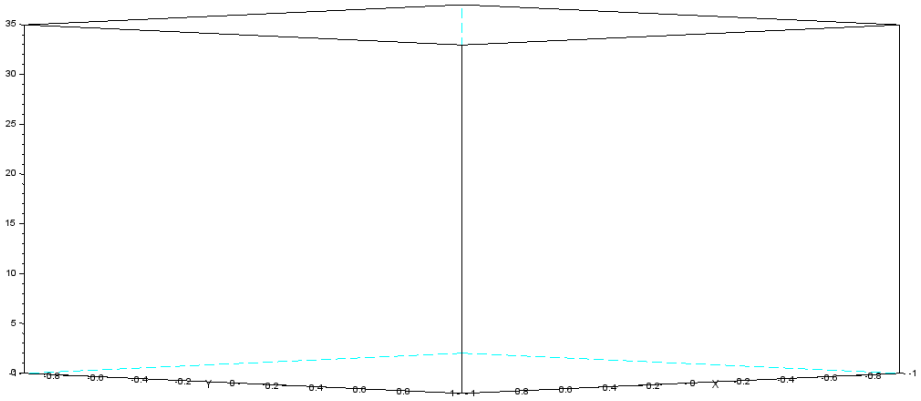

FIGURE 4.3 – Représentation d'une spirale en 3D.

La fonction `surf` permet de créer des surfaces en 3D à partir de données discrètes. Par exemple, pour représenter une fonction $z = f(x, y)$, vous pouvez utiliser la commande suivante :

```
x = linspace(-10, 10, 100);
y = linspace(-5, 5, 100);
[X, Y] = meshgrid(x, y);
Z = X.^3 + Y.^4;
surf(X, Y, Z);
```

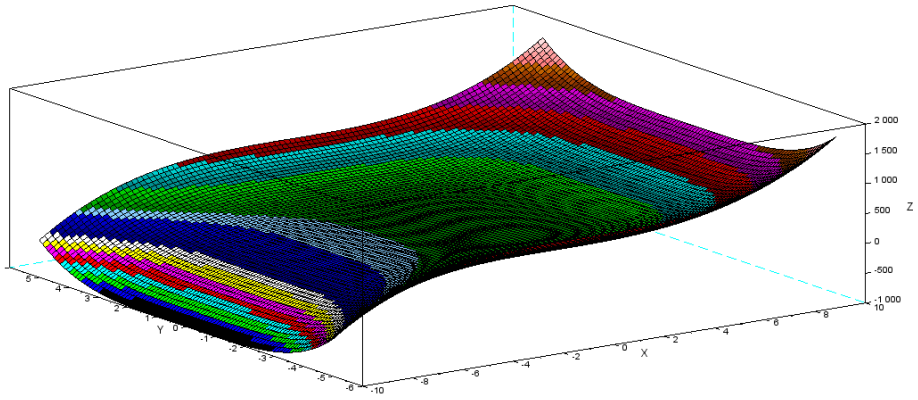

FIGURE 4.4 – Création de surfaces en 3D à partir de la fonction $z = f(x, y)$.

Scilab offre également des fonctions pour créer des graphiques de type contour, des graphiques de type nuage de points en 3D, etc. Vous pouvez consulter la documentation de Scilab pour en savoir plus sur ces différentes fonctions de visualisation.

4.1.2 Personnalisation des graphiques

Scilab offre de nombreuses options de personnalisation pour les graphiques. Vous pouvez modifier les couleurs, les styles de ligne, les légendes, les étiquettes des axes, etc. pour rendre vos graphiques plus attrayants et informatifs.

Par exemple, vous pouvez utiliser la fonction `xlabel` pour ajouter une étiquette à l'axe des abscisses, la fonction `ylabel` pour ajouter une étiquette à l'axe des ordonnées, et la fonction `title` pour ajouter un titre à votre graphique. Vous pouvez également utiliser les fonctions `grid` et `legend` pour ajouter une grille et une légende à votre graphique, respectivement.

```
x = linspace(0, 2*%pi, 100);
y = sin(x);
plot(x, y);
xlabel('x');
ylabel('sin(x)');
title('Graphique de la fonction sin(x)');
grid on;
legend('sin(x)');
```

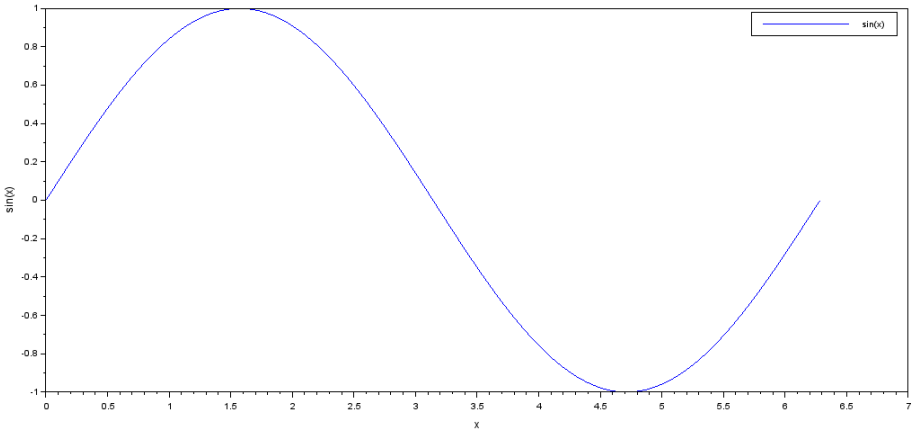

FIGURE 4.5 – Graphique de la fonction sin(x)

Pour changer la couleur d'un graphique en 3D, nous utilisons l'option `color`. Par exemple, pour définir la couleur du graphique en 3D en rouge, nous pouvons utiliser le code suivant :

```
// Données pour le graphe
x = 1:10;
y = x.^2; // Carré des valeurs de x
// Tracer le graphe avec la couleur rouge
plot(x, y, 'r'); // 'r' spécifie la couleur rouge
```

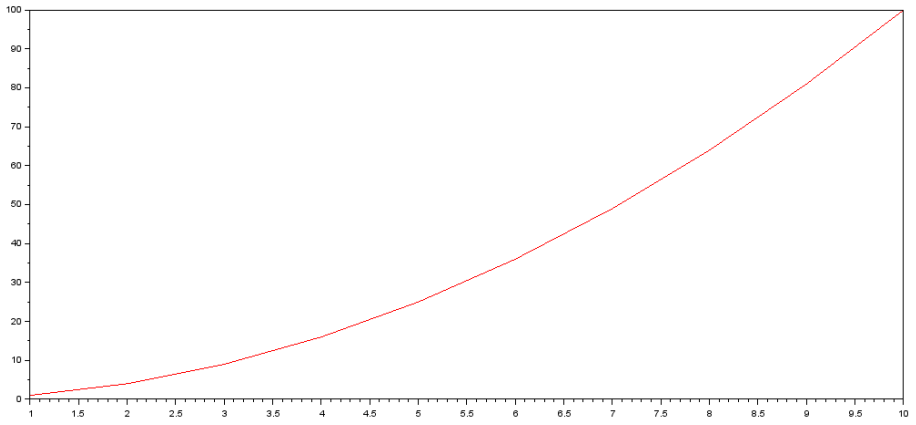

FIGURE 4.6 – Graphique de la fonction x^2

4.2 Les animations

Les animations sont un outil puissant pour visualiser et comprendre les phénomènes mathématiques en mouvement. Dans Scilab, il est possible de créer des animations en utilisant différentes techniques et fonctions. Dans cette section, nous explorerons les différentes méthodes pour créer des animations dans Scilab.

4.2.1 Animation basée sur des graphiques

Une des façons les plus courantes de créer des animations dans Scilab est d'utiliser des graphiques. Scilab offre de nombreuses fonctions graphiques qui permettent de créer des graphiques en mouvement. Par exemple, la fonction `plot2d` peut être utilisée pour tracer des courbes en temps réel. En utilisant une boucle, il est possible de mettre à jour les données du graphique à chaque itération, créant ainsi une animation fluide.

Voici un exemple simple qui montre comment créer une animation basée sur un graphique en utilisant la fonction `plot2d` :

```
// Création d'un vecteur temps
t = 0:0.1:10;
// Création d'un vecteur de données
x = sin(t);
// Tracer le graphique initial
xtitle('Animation basée sur un graphique','Temps', 'Amplitude')
h = plot2d(t, x, style = 1);

// Ajouter des titres aux axes
for i = 1:length(t)
    x_partial = x(1:i);
    // Extraire les données jusqu'à l'itération actuelle
    t_partial = t(1:i);
    // Correspondant au vecteur temps partiel
    set(h, 'xdata', t_partial, 'ydata', x_partial);
    // Mettre à jour les données affichées
    pause(0.1) // Pause pour créer l'effet d'animation
end
```

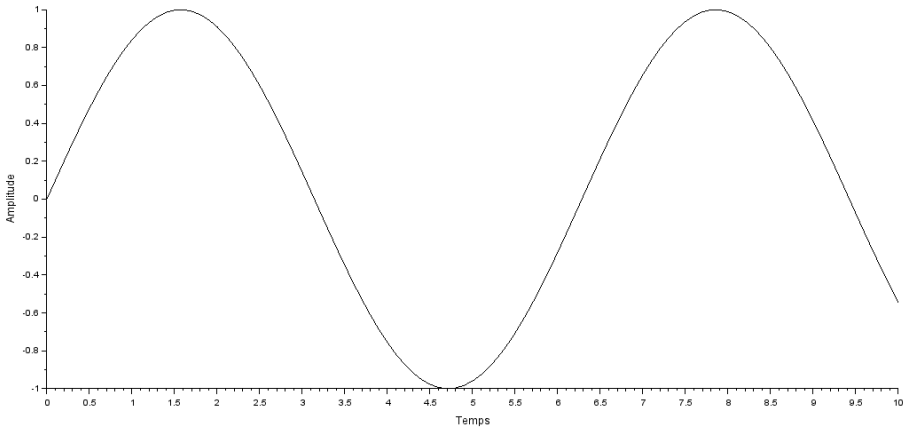

FIGURE 4.7 – Animation basée sur un graphique en utilisant la fonction plot2d pour la fonction sin(t)

Remarque

Ce code crée un graphique initial d'une sinusoïde puis, dans une boucle, met à jour progressivement le graphique pour afficher les données jusqu'à l'itération actuelle, créant ainsi une animation montrant l'évolution de la sinusoïde au fil du temps.

4.2.2 Animation basée sur des transformations géométriques

Une autre façon de créer des animations dans Scilab est d'utiliser des transformations géométriques. Scilab offre des fonctions pour effectuer des translations, des rotations et des mises à l'échelle sur des objets graphiques. En utilisant ces fonctions dans une boucle, il est possible de créer des animations qui montrent le mouvement d'objets dans l'espace.

Voici un exemple qui montre comment créer une animation basée sur des transformations géométriques :

```
clf // Effacer la fenêtre graphique

x = zeros(1, 100); // Initialiser un vecteur de coordonnées x
y = zeros(1, 100); // Initialiser un vecteur de coordonnées y
z = zeros(1, 100); // Initialiser un vecteur de coordonnées z

h = plot3d(x, y, z, 'style', 1, 'mark', 8);

```

```
// Créer le graphique initial avec des vecteurs de coordonnées vides

xtitle('Animation basée sur des transformations géométriques', 'X',
↪  'Y', 'Z');

// Ajouter des titres aux axes
for i = 1:100
    // Coordonnée x de l'objet (ajustement de l'échelle)
    x(i) = i * 0.1;
    // Coordonnée y de l'objet (ajustement de l'échelle)
    y(i) = i * 0.1;
    // Coordonnée z de l'objet (ajustement de l'échelle)
    z(i) = i * 0.1;

    // Mettre à jour les données de l'objet sur le graphique
    plot3d(x(1:i), y(1:i), z(1:i), style = 1, mark = 8);

    // Pause pour créer l'effet d'animation
    pause(0.1)
end
```

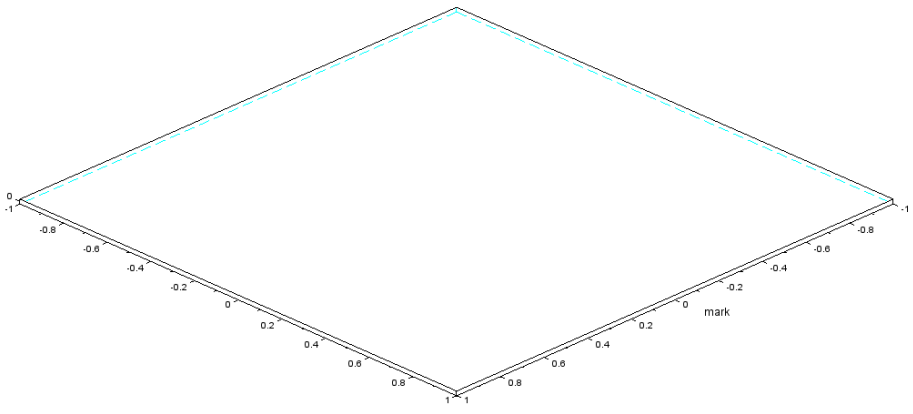

FIGURE 4.8 – Animation basée sur des transformations géométriques

Remarque

Ce code génère une animation en déplaçant un objet dans l'espace 3D en mettant à jour ses coordonnées à chaque itération de la boucle, offrant ainsi une visualisation d'une trajectoire ou d'un mouvement dans l'espace.

4.2.3 Animation basée sur des équations mathématiques

Scilab permet également de créer des animations basées sur des équations mathématiques. En utilisant des fonctions mathématiques et des boucles, il est possible de créer des animations qui montrent l'évolution de différentes fonctions au fil du temps.

```
clf(); // Effacer la fenêtre graphique

t = 0:0.1:10;
h = plot2d(t, sin(t), style = 1);

// Tracer la fonction initiale
xtitle('Animation basée sur une équation mathématique','X','Y')

// Ajouter des titres aux axes
for i = 1:length(t)
    y = sin(t(i)*t(i));
    // Calcul de la fonction avec t(i) comme variable indépendante
    if i == 1
        clf(); // Effacer la fenêtre graphique avant de tracer la
        ↪  première itération
        h = plot2d(t(1:i), sin(t(1:i).*t(1:i)), style = 1, mark = 8);
        // Tracer la première itération
        xtitle('Animation basée sur une équation mathématique','X','Y')
        // Ajouter des titres aux axes
    else
        h.children(1).children(1).data = [t(1:i); sin(t(1:i).*t(1:i))];
        // Modifier les données du graphique
        h.children(1).children(1).polyline_mode = "no";
        // Actualiser le mode de traçage
    end
    pause(0.1)
    // Pause pour créer l'effet d'animation
end
```

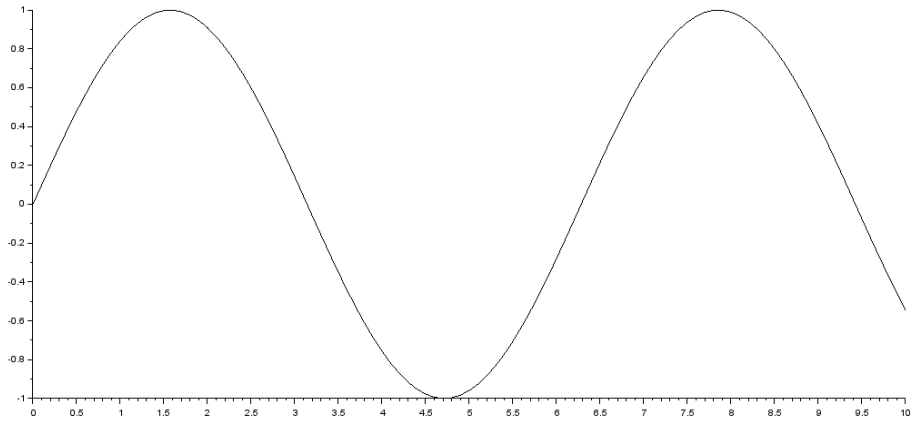

FIGURE 4.9 – Animation basée sur des équations mathématiques

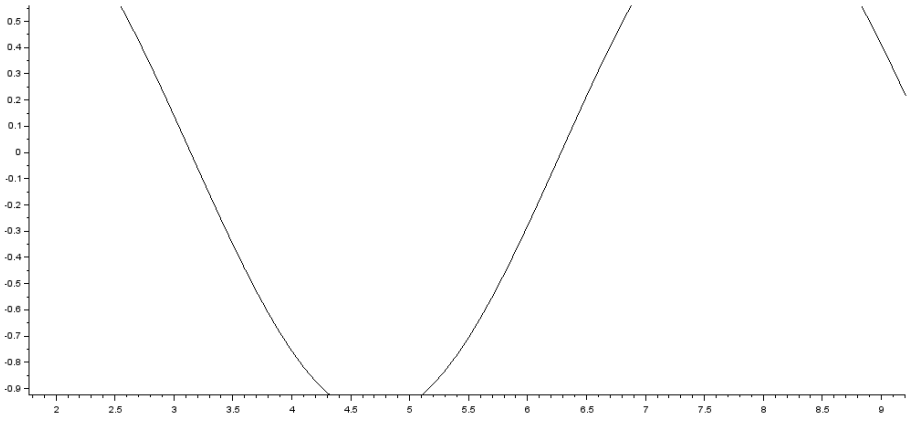

FIGURE 4.10 – Animation basée sur des équations mathématiques

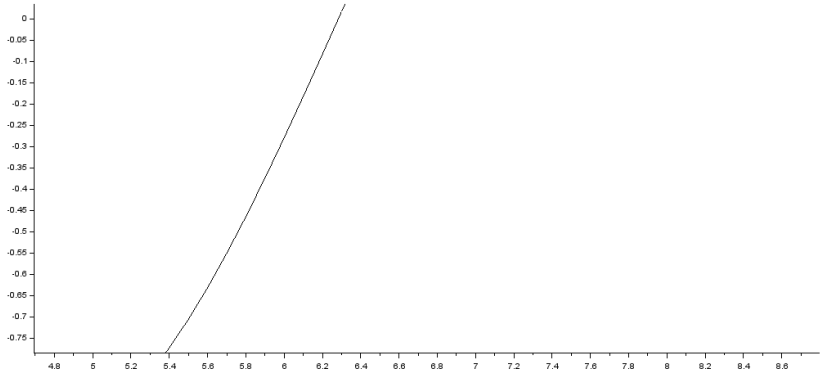

FIGURE 4.11 – Animation basée sur des équations mathématiques

Remarque

Ce code crée une animation graphique en affichant une fonction sinusoïdale qui évolue au fil du temps, démontrant comment la fonction varie lorsque la variable indépendante t est modifiée selon $t(i) \cdot t(i)$ à chaque étape.

4.3 Les diagrammes

Les diagrammes sont des outils visuels puissants pour représenter des données de manière claire et concise. Dans Scilab, il existe plusieurs types de diagrammes que vous pouvez créer pour visualiser vos données. Dans cette section, nous explorerons les différents types de diagrammes disponibles dans Scilab et comment les créer.

4.3.1 Diagrammes en barres

Les diagrammes en barres sont utilisés pour représenter des données catégorielles. Ils sont particulièrement utiles pour comparer des valeurs entre différentes catégories. Dans Scilab, vous pouvez créer des diagrammes en barres en utilisant la fonction barplot. Cette fonction prend en entrée un vecteur de données et génère un diagramme en barres correspondant.

```
x = [1, 2, 3, 4, 5];
y = [10, 15, 7, 12, 8];
bar(x, y);
xtitle('Diagramme à barres', 'X', 'Y'); // Ajouter des titres aux axes
```

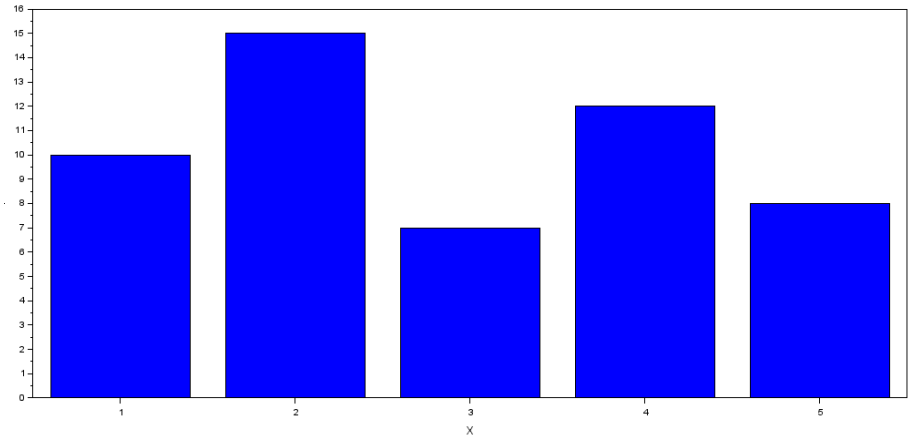

FIGURE 4.12 – Diagramme à barres

4.3.2 Diagrammes circulaires

Les diagrammes circulaires, également appelés diagrammes en secteurs, sont utilisés pour représenter des proportions relatives. Ils sont souvent utilisés pour visualiser la répartition d'une variable catégorielle en pourcentages. Dans Scilab, vous pouvez créer des diagrammes circulaires en utilisant la fonction pieplot. Cette fonction prend en entrée un vecteur de données et génère un diagramme circulaire correspondant.

```
// Données
labels = ['A', 'B', 'C', 'D', 'E'];
valeurs = [10, 15, 7, 12, 8];

// Créer le diagramme circulaire
pie(valeurs, labels);
title('Diagramme circulaire');

// Afficher la légende
legend(labels);
```

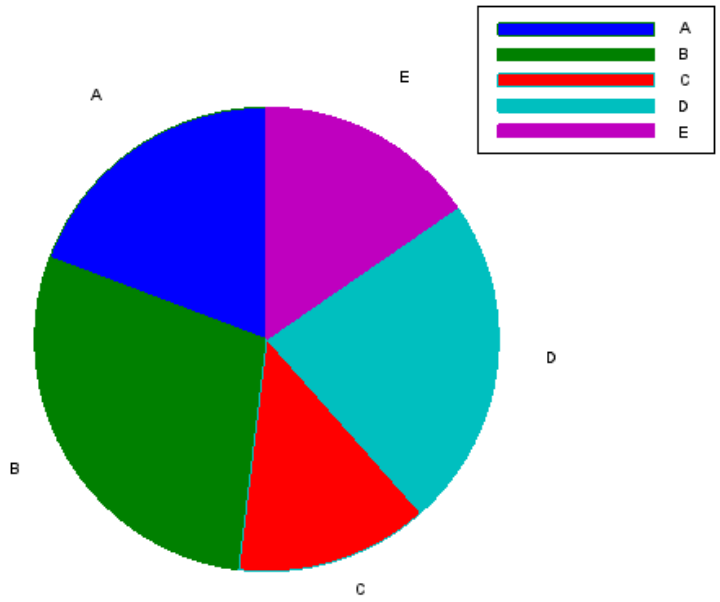

FIGURE 4.13 – Diagramme circulaire

4.3.3 Diagrammes en nuages de points

Les diagrammes en nuages de points sont utilisés pour représenter la relation entre deux variables continues. Ils sont particulièrement utiles pour détecter des tendances ou des corrélations entre les variables. Dans Scilab, vous pouvez créer des diagrammes en nuages de points en utilisant la fonction scatterplot. Cette fonction prend en entrée deux vecteurs de données, un pour chaque variable, et génère un diagramme en nuages de points correspondant.

```
x = [1, 4, 8, 10, 12];
y = [10, 15, 7, 12, 8];

// Utilisation du marqueur 'O' pour indiquer les points sous forme de
↪  cercles
plot(x, y, 'o');

// Ajouter des titres aux axes
xtitle('Nuage de points', 'X', 'Y');
```

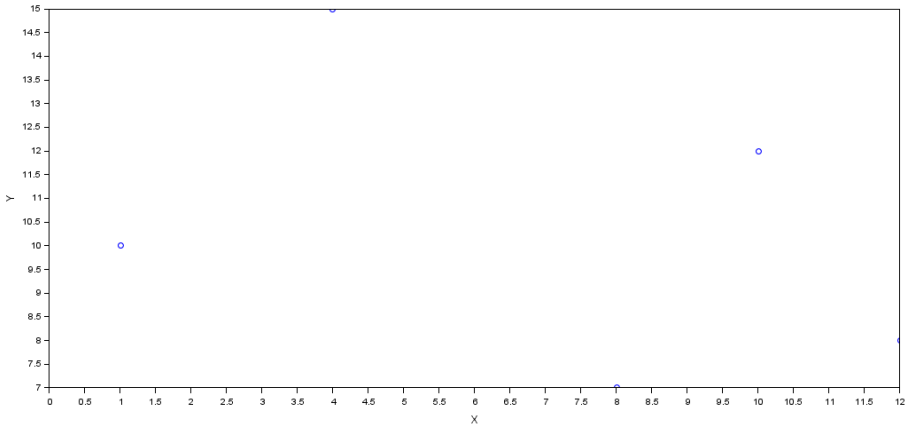

FIGURE 4.14 – Diagrammes en nuages de points

4.3.4 Diagrammes en boîte

Les diagrammes en boîte, également appelés diagrammes de dispersion, sont utilisés pour représenter la distribution d'un ensemble de données. Ils fournissent des informations sur la médiane, les quartiles et les valeurs aberrantes des données. Dans Scilab, vous pouvez créer des diagrammes en boîte en utilisant la fonction boxplot. Cette fonction prend en entrée un vecteur de données et génère un diagramme en boîte correspondant.

```
// Calcul des statistiques de base
x = [10, 15, 7, 12, 8];
median_value = median(x);
// Premier quartile
q1 = quantile(x, 0.25);
// Troisième quartile
q3 = quantile(x, 0.75);
min_value = min(x);
max_value = max(x);

// Tracé du diagramme en boîte à moustaches
boxplot_data = [min_value q1 median_value q3 max_value];

// Tracer les points représentant les statistiques
boxplot_x = ones(1, 5);

// Tracer les lignes pour la boîte
```

```
plot(boxplot_x, boxplot_data, 'r+');

// Premier quartile
plot([0.9, 1.1], [q1, q1], '-b');

// Troisième quartile
plot([0.9, 1.1], [q3, q3], '-b');

// Médiane
plot([0.9, 1.1], [median_value, median_value], '-g');

// Valeur minimale
plot([0.85, 1.15], [min_value, min_value], '-r');

// Valeur maximale
plot([0.85, 1.15], [max_value, max_value], '-r');

// Masquer les marques sur l'axe x
xset('xtick', []);

// Ajouter un titre à l'axe x
xtitle('Diagramme en boîte à moustaches', 'Données');
```

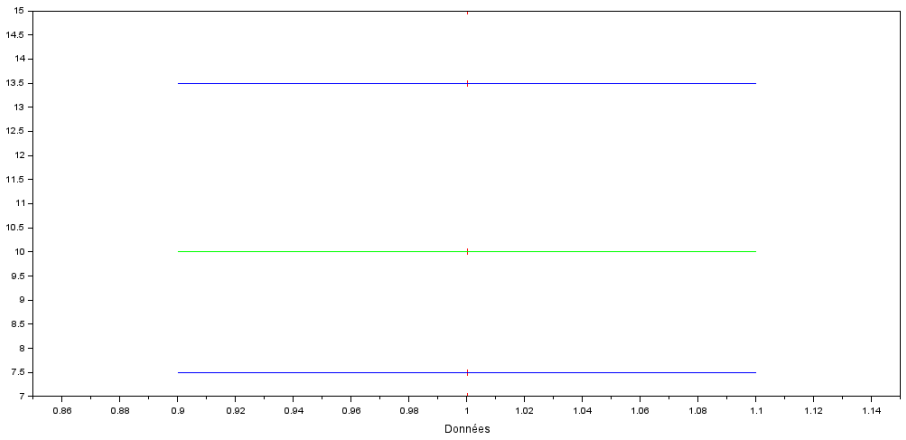

FIGURE 4.15 – Diagrammes en boîte

Travaux Pratique 4 : Graphiques et visualisation

Graphique de Fonction

— Tracez la fonction $y = x^2$ sur l'intervalle $[0, 5]$.
— Ajoutez une grille au graphique.
— Ajoutez des étiquettes aux axes et un titre au graphique.

Diagramme en Barres

— Générez un vecteur de données représentant les ventes mensuelles d'un produit.
ventes mensuelles = [120, 90, 150, 80, 110, 130] ;
— Créez un diagramme en barres pour visualiser ces données.
— Ajoutez différentes couleurs pour chaque barre et une légende.

Nuage de Points et Régression Linéaire

— Générez deux vecteurs de données x et y pour un nuage de points.
— Tracez un nuage de points avec x sur l'axe des abscisses et y sur l'axe des ordonnées.
— Ajoutez une ligne de régression linéaire au graphique.

Graphique en 3D

— Générez des données pour une fonction tridimensionnelle, par exemple $z = \sin(x) + \cos(y)$.
— Tracez un graphique en 3D de cette fonction.
— Ajoutez des étiquettes aux axes et un titre.

Diagramme Circulaire

— Générez un vecteur représentant la répartition des dépenses mensuelles.
— Créez un diagramme circulaire pour visualiser cette répartition.
— Ajoutez des pourcentages sur chaque portion du diagramme.

Sous-Graphiques

— Tracez plusieurs graphiques sur une même figure à l'aide de sous-graphiques.
— Utilisez `subplot` pour organiser les graphiques de manière appropriée.

Correction de TP 4 : Graphiques et visualisation

Graphique de Fonction

```
// Trace la fonction y = x^2 sur l'intervalle [0, 5]
x = linspace(0, 5, 100);
y = x.^2;

// Plot la fonction
plot(x, y, 'b-', 'LineWidth', 2);

// Ajoute une grille au graphique
grid on;

// Ajoute des étiquettes aux axes et un titre
xlabel('Axe des abscisses');
ylabel('Axe des ordonnées');
title('Graphique de la fonction y = x^2');

// Affiche la légende
legend('y = x^2', 'Location', 'NorthWest');
```

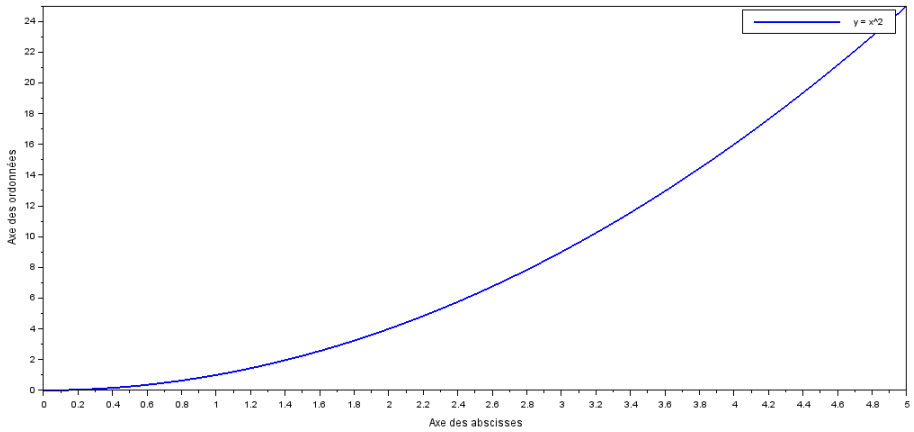

FIGURE 4.16 – Graphique de la fonction $y = x^2$

Diagramme en Barres

```
// Génère un vecteur de données représentant les ventes mensuelles d'un produit
ventes_mensuelles = [120, 90, 150, 80, 110, 130];

// Crée un diagramme en barres pour visualiser ces données
bar(ventes_mensuelles, 'b', 'BarWidth', 0.6);

// Ajoute des couleurs différentes pour chaque barre
colors = jet(length(ventes_mensuelles));
for i = 1:length(ventes_mensuelles)
    h = findobj(gca(), 'Type', 'Bar');
    h(i).FaceColor = colors(i,:);
end

// Ajoute une légende
legend(cellstr(['Mois ', string(1:length(ventes_mensuelles))]), 'Location', 'NorthEast');

// Ajoute des étiquettes aux axes et un titre
xlabel('Mois');
ylabel('Ventes Mensuelles');
title('Diagramme en Barres des Ventes Mensuelles');
```

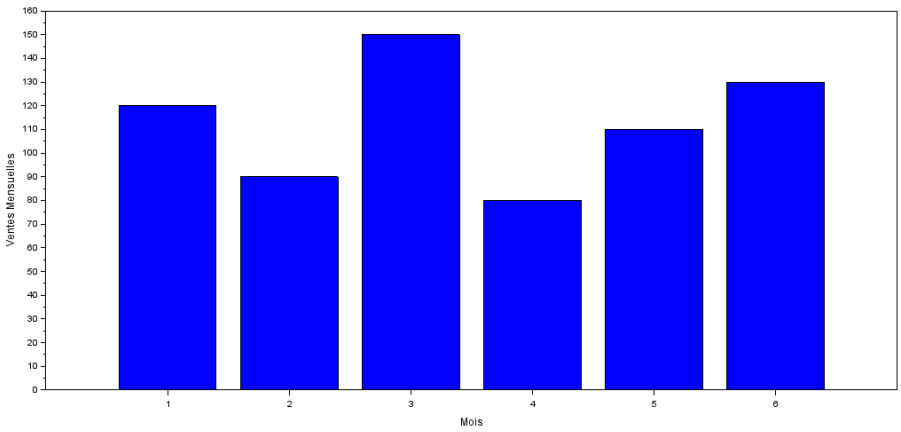

FIGURE 4.17 – Diagramme en Barres des Ventes Mensuelles.

Nuage de Points et Régression Linéaire

```
// Génération de données aléatoires pour x
x = rand(1, 50);

// Relation linéaire avec bruit pour y
y = 2 * x + rand(1, 50) * 0.1;

// Tracer le nuage de points
clf;
scatter(x, y, 'filled');
xlabel('x');
ylabel('y');
title('Nuage de points avec régression linéaire');

// Coefficients de la régression linéaire
coeff = polyfit(x, y, 1);

// Calcul de la régression linéaire
line = coeff(1) * x + coeff(2);

// Tracer la ligne de régression linéaire
plot(x, line, 'r');

// Afficher le graphique
legend('Données','Régression linéaire','location','northwest');
```

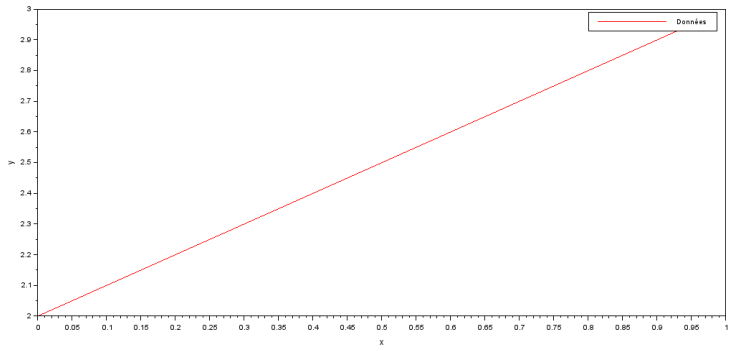

FIGURE 4.18 – Nuage de points avec régression linéaire

Graphique en 3D

```
// Définir la grille de points pour x et y
x = linspace(0, 2*%pi, 100);
y = linspace(0, 2*%pi, 100);
[x, y] = ndgrid(x, y);

// Calculer les valeurs de z
z = sin(x) + cos(y);

// Tracer le graphique en 3D
clf;
plot3d(x, y, z);
xlabel('X');
ylabel('Y');
zlabel('Z');
title('Graphique en 3D de z = sin(x) + cos(y)');
```

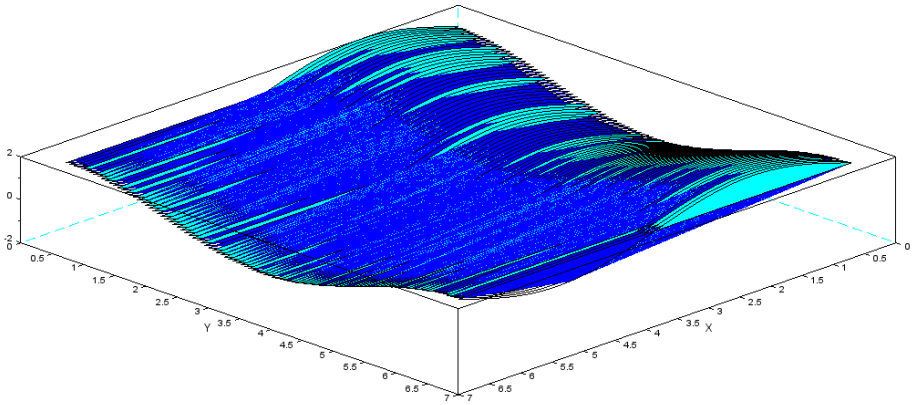

FIGURE 4.19 – Graphique en 3D de z = sin(x) + cos(y)

Diagramme Circulaire

```
// Générer un vecteur représentant la répartition des dépenses
mensuelles
depenses = [2000, 1500, 1200, 800, 600];
// Exemple de dépenses dans l'ordre: alimentation, loyer,
transport, divertissement, autres

// Créer un diagramme circulaire pour visualiser cette répartition
clf;
pie(depenses);
legend('Alimentation', 'Loyer', 'Transport', 'Divertissement',
'Autres');
// Ajout de la légende correspondant à chaque portion
// du diagramme

// Ajouter un titre au graphique
title('Répartition des dépenses mensuelles');

// Ajouter des pourcentages sur chaque portion du diagramme
percentage = round((depenses / sum(depenses)) * 100);
// Calculer les pourcentages arrondis
```

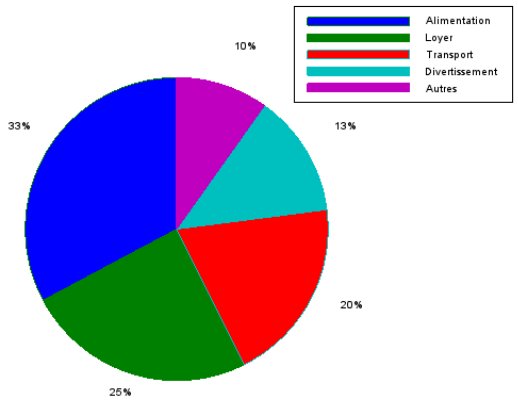

FIGURE 4.20 – Répartition des dépenses mensuelles

Sous-graphiques

```
// Générer des données pour les graphiques
x = 0:0.1:2*%pi;
y1 = sin(x);
y2 = cos(x);
y3 = sin(2*x);
y4 = cos(2*x);

clf; // Créer une nouvelle figure
// Utiliser subplot pour organiser les graphiques de manière appropriée

subplot(2, 2, 1); // Sous-graphique 1
plot(x, y1);
title('sin(x)');

subplot(2, 2, 2); // Sous-graphique 2
plot(x, y2);
title('cos(x)');

subplot(2, 2, 3); // Sous-graphique 3
plot(x, y3);
title('sin(2x)');

subplot(2, 2, 4); // Sous-graphique 4
plot(x, y4);
title('cos(2x)');
```

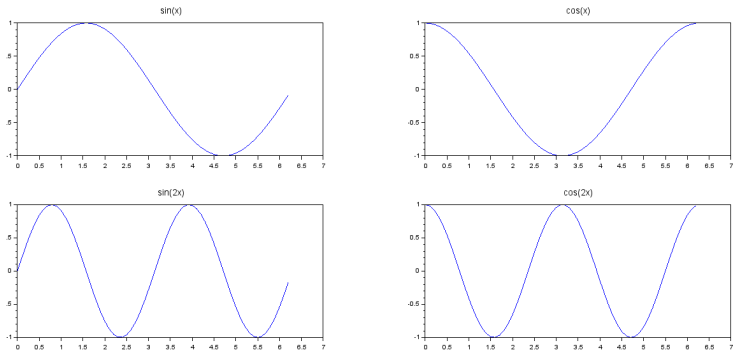

FIGURE 4.21 – Évolution périodique des fonctions sinus et cosinus

Chapitre 5

Calculs numériques

5.1 Les méthodes numériques

Les méthodes numériques sont des techniques utilisées pour résoudre des problèmes mathématiques complexes qui ne peuvent pas être résolus analytiquement. Elles sont particulièrement utiles dans les domaines de l'ingénierie, de la physique et des sciences appliquées. Scilab offre une large gamme de méthodes numériques qui permettent de résoudre efficacement ces problèmes.

Par exemple, la fonction fsolve permet de résoudre numériquement des équations non linéaires. Elle utilise une méthode itérative pour trouver une solution approchée à l'équation donnée. La syntaxe de la fonction est la suivante

```
x = fsolve(f, x0)
```

Où f est la fonction à résoudre et x0 est une estimation initiale de la solution. La fonction renvoie la valeur de x qui satisfait l'équation donnée.

Exemple Supposons que nous avons une équation $f(x) = x^2 - 4 = 0$ Nous allons utiliser fsolve pour trouver la valeur de x qui satisfait cette équation.

```
// Définition de la fonction à résoudre
function y = my_function(x)
    y = x^2 - 4;
endfunction

// Valeur initiale pour x
x0 = 2;

```

```
// Utilisation de fsolve pour trouver la solution
x = fsolve(my_function, x0);

// Affichage de la solution
disp(x);
```

5.2 Intégrations numériques

Les intégrations numériques sont des méthodes utilisées pour calculer une approximation numérique de l'intégrale d'une fonction sur un intervalle donné. Ces méthodes sont particulièrement utiles lorsque l'intégrale ne peut pas être calculée analytiquement ou lorsque la fonction est complexe.

Exemple La méthode des trapèzes pour estimer l'intégrale de $f(x) = x^2$ sur l'intervalle [0,1].

```
// Définition de la fonction à intégrer
function y = my_function(x)
    y = x^2;
endfunction

// Bornes de l'intervalle d'intégration
a = 0;
b = 1;

// Nombre de sous-intervalles
n = 100;

// Calcul de l'intégrale numérique avec la méthode des trapèzes
// Largeur de chaque sous-intervalle
h = (b - a) / n;

// Points d'échantillonnage
x = a:h:b;

// Valeurs de la fonction aux points d'échantillonnage
y = my_function(x);

// Formule de la méthode des trapèzes
```

```
I = h * (sum(y) - (y(1) + y(n+1)) / 2);

// Affichage du résultat
disp(I);
```

5.3 Les résolutions d'équations

La résolution d'équations est une tâche courante en mathématiques et en sciences. Scilab offre de puissants outils pour résoudre différents types d'équations, qu'elles soient linéaires ou non linéaires. Dans cette section, nous explorerons les différentes méthodes disponibles dans Scilab pour résoudre des équations.

5.3.1 Résolution d'équations linéaires

Les équations linéaires sont des équations de la forme Ax = b, où A est une matrice, x est un vecteur inconnu et b est un vecteur constant.

Pour résoudre des équations linéaires en Scilab, vous pouvez utiliser la fonction linsolve ou la notation "/" pour résoudre des systèmes linéaires.

Exemple

— Utilisation de linsolve : La fonction linsolve est utilisée pour résoudre des systèmes d'équations linéaires. comme suit :

```
// Matrice des coefficients
A = [2, 1, -1; 1, -1, 1; 3, -2, 1];
// Vecteur des termes constants
b = [2; 1; 4];
// Résolution du système d'équations linéaires
x = linsolve(A, b);
// Affichage de la solution
disp(x);
```

— Utilisation de la notation Scilab permet également d'utiliser la notation pour résoudre des systèmes linéaires en utilisant l'opérateur de division gauche. comme suit

```
// Matrice des coefficients
A = [2, 1, -1; 1, -1, 1; 3, -2, 1];
```

```
// Vecteur des termes constants
b = [2; 1; 4];
// Résolution du système d'équations linéaires
x = A \ b;
// Affichage de la solution
disp(x);
```

5.3.1.1 Élimination de Gauss

L'élimination de Gauss est une méthode directe pour résoudre des systèmes d'équations linéaires en les transformant en une forme échelonnée. Elle consiste à appliquer des opérations élémentaires sur les lignes de la matrice augmentée du système jusqu'à ce qu'elle soit sous forme échelonnée. Ensuite, les solutions sont obtenues par substitution arrière.

```
function x = gauss_elimination(A, b)
    n = size(A, 1);
    Ab = [A, b]; // Matrice augmentée

    // Étape d'élimination
    for k = 1:n-1
        for i = k+1:n
            coef = Ab(i,k) / Ab(k,k);
            Ab(i,:) = Ab(i,:) - coef * Ab(k,:);
        end
    end

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

// Exemple d'utilisation
A = [2 1 -1; -3 -1 2; -2 1 2];
b = [8; -11; -3];
x = gauss_elimination(A, b);
disp(x);
```

5.3.1.2 Décomposition LU

La décomposition LU consiste à décomposer la matrice du système en un produit de deux matrices : une matrice triangulaire inférieure (L) et une matrice triangulaire supérieure (U). Ensuite, le système est résolu en résolvant deux systèmes linéaires triangulaires.

```
function x = lu_decomposition_solve(A, b)
    [L, U] = lu(A);
    y = L\b;
    x = U\y;
end

// Exemple d'utilisation
A = [2 1 -1; -3 -1 2; -2 1 2];
b = [8; -11; -3];
x = lu_decomposition_solve(A, b);
disp(x);
```

5.3.1.3 Méthode de Jacobi

La méthode de Jacobi est une méthode itérative qui consiste à itérer sur les composantes de la solution jusqu'à ce qu'une certaine condition de convergence soit atteinte.

```
function x = jacobi(A, b, tol, max_iter)
    n = length(b);
    x = zeros(n, 1); // Initialisation de la solution
    x_old = zeros(n, 1);
    iter = 0;

    while iter < max_iter
        x_old = x;
        for i = 1:n
            sigma = 0;
            for j = 1:n
                if j != i
                    sigma = sigma + A(i, j) * x_old(j);
                end
            end
            x(i) = (b(i) - sigma) / A(i, i);
```

```
        end
        if norm(x - x_old, "inf") < tol
            break;
        end
        iter = iter + 1;
    end
end

// Exemple d'utilisation
A = [4 -1 0; -1 4 -1; 0 -1 4];
b = [10; 10; 10];
tol = 1e-6; // Tolérance de convergence
max_iter = 1000; // Nombre maximal d'itérations
x = jacobi(A, b, tol, max_iter);
disp(x);
```

5.3.1.4 Méthode de Gauss-Seidel

La méthode de Gauss-Seidel est une amélioration de la méthode de Jacobi qui utilise les valeurs mises à jour dès qu'elles sont disponibles au lieu d'attendre la fin de chaque itération. Cela peut conduire à une convergence plus rapide, en particulier pour les systèmes bien conditionnés.

```
function x = gauss_seidel(A, b, tol, max_iter)
    n = length(b);
    x = zeros(n, 1); // Initialisation de la solution
    x_old = zeros(n, 1);
    iter = 0;

    while iter < max_iter
        x_old = x;
        for i = 1:n
            sigma = 0;
            for j = 1:n
                if j != i
                    sigma = sigma + A(i, j) * x(j);
                end
            end
            x(i) = (b(i) - sigma) / A(i, i);
        end
```

```
        if norm(x - x_old, "inf") < tol
            break;
        end
        iter = iter + 1;
    end
end

// Exemple d'utilisation
A = [4 -1 0; -1 4 -1; 0 -1 4];
b = [10; 10; 10];
tol = 1e-6; // Tolérance de convergence
max_iter = 1000; // Nombre maximal d'itérations
x = gauss_seidel(A, b, tol, max_iter);
disp(x);
```

5.3.1.5 Méthode de relaxation

La méthode de relaxation est une extension de la méthode de Gauss-Seidel qui introduit un paramètre de relaxation pour accélérer la convergence. Ce paramètre permet de contrôler la vitesse de convergence en ajustant la pondération des valeurs mises à jour.

```
function x = relaxation(A, b, omega, tol, max_iter)
    n = length(b);
    x = zeros(n, 1); // Initialisation de la solution
    x_old = zeros(n, 1);
    iter = 0;

    while iter < max_iter
        x_old = x;
        for i = 1:n
            sigma = 0;
            for j = 1:n
                if j != i
                    sigma = sigma + A(i, j) * x(j);
                end
            end
            x(i) = (1 - omega) * x_old(i) + (omega / A(i, i))
            * (b(i) - sigma);
        end
```

```
        if norm(x - x_old, "inf") < tol
            break;
        end
        iter = iter + 1;
    end
end

// Exemple d'utilisation
A = [4 -1 0; -1 4 -1; 0 -1 4];
b = [10; 10; 10];
omega = 1.2; // Facteur de sur-relaxation
tol = 1e-6; // Tolérance de convergence
max_iter = 1000; // Nombre maximal d'itérations
x = relaxation(A, b, omega, tol, max_iter);
disp(x);
```

5.3.1.6 Méthode des moindres carrés

La méthode des moindres carrés est utilisée pour résoudre les systèmes surdéterminés, c'est-à-dire lorsqu'il y a plus d'équations que d'inconnues. Elle trouve la solution qui minimise la somme des carrés des résidus entre les valeurs prédites et les valeurs réelles. Cette méthode est largement utilisée en régression linéaire et dans d'autres applications statistiques.

```
function x = least_squares(A, b)
    x = pinv(A) * b; // Solution par la pseudo-inverse de A
end

// Exemple d'utilisation
A = [1 2; 3 4; 5 6];
b = [5; 11; 17];
x = least_squares(A, b);
disp(x);
```

5.3.2 Résolution d'équations non linéaires

Les équations non linéaires sont des équations dans lesquelles les variables inconnues apparaissent dans des termes non linéaires. Résoudre ces équations peut être plus complexe que résoudre des équations linéaires, car il n'existe pas de méthode générale pour trouver une solution exacte

Pour résoudre des équations non linéaires sur Scilab, vous pouvez utiliser différentes méthodes numériques, telles que la méthode de Newton-Raphson, la méthode de la sécante, la méthode de la bissection, entre autres. Ces méthodes visent à trouver les solutions d'équations non linéaires en utilisant des itérations successives.

5.3.2.1 Méthode de Newton-Raphson

La méthode de Newton-Raphson est une méthode itérative qui converge rapidement vers la solution d'une équation en utilisant les dérivées de la fonction. Voici un exemple de résolution d'une équation non linéaire $f(x) = 0$ avec la méthode de Newton-Raphson

```
function [x, iterations] = newton_raphson(func, df, x0, tolerance,
↪  max_iterations)
    iterations = 0;
    x = x0;

    while abs(feval(func, x)) > tolerance &&
    iterations < max_iterations
        x = x - feval(func, x) / feval(df, x);
        iterations = iterations + 1;
    end
endfunction

// Exemple d'utilisation avec une fonction et sa dérivée
function y = my_function(x)
    y = x^2 - 4; // équation à résoudre: x^2 - 4 = 0
endfunction

function dy = my_derivative(x)
    dy = 2 * x; // dérivée de la fonction x^2 - 4
endfunction

x0 = 2; // estimation initiale
tolerance = 1e-6;
max_iterations = 100;

[x, iterations] =
newton_raphson('my_function', 'my_derivative', x0, tolerance,
max_iterations);
disp(x);
```

Remarque

Scilab propose également d'autres méthodes numériques pour résoudre des équations non linéaires, telles que fsolve, qui peut être utilisée de manière similaire à la méthode de Newton-Raphson pour résoudre des équations non linéaires.

```
// Fonction dont la racine est recherchée
function y = my_function(x)
    y = x^2 - 4; // équation à résoudre: x^2 - 4 = 0
endfunction

// Estimation initiale
x0 = 2;

// Résolution de l'équation non linéaire avec fsolve
x_solution = fsolve(x0, my_function);

// Affichage de la solution
disp(x_solution);
```

5.3.2.2 Méthode de la dichotomie

Cette méthode est basée sur le théorème des valeurs intermédiaires et est utilisée pour trouver une solution d'une équation non linéaire sur un intervalle donné. Elle consiste à diviser l'intervalle en deux parties et à déterminer dans quelle partie se trouve la racine de l'équation. En répétant ce processus de division et d'évaluation de l'équation sur des sous-intervalles, on peut converger vers une approximation de la racine.

```
// Définition de la fonction à résoudre
function y = f(x)
    y = x^2 - 2;
endfunction

function x = dichotomy_method(f, a, b, tol, max_iter)
    if f(a) * f(b) > 0 then
        error('Pas de changement de signe dans l''intervalle
        donné.');
    end
```

```

    iter = 0;
    while iter < max_iter
        c = (a + b) / 2;
        if f(c) == 0 || (b - a) / 2 < tol then
            x = c;
            return;
        end
        if f(c) * f(a) < 0 then
            b = c;
        else
            a = c;
        end
        iter = iter + 1;
    end
    error('La méthode de dichotomie a échoué pour converger.');
endfunction

// Exemple d'utilisation
a = 0;
b = 2;
tol = 1e-6;
max_iter = 1000;
x = dichotomy_method(f, a, b, tol, max_iter);
disp(x);
```

5.3.2.3 Méthode de la sécante :

La méthode de la sécante est une variante de la méthode de Newton qui n'exige pas le calcul de la dérivée de la fonction. Au lieu de cela, elle utilise une approximation de la dérivée basée sur deux points successifs de la fonction. Bien que cette méthode puisse converger moins rapidement que la méthode de Newton, elle reste efficace dans de nombreux cas.

```
function x = secant_method(f, x0, x1, tol, max_iter)
    iter = 0;
    while iter < max_iter
        x = x1 - f(x1) * (x1 - x0) / (f(x1) - f(x0));
        if abs(x - x1) < tol then
            return;
```

```
        end
        x0 = x1;
        x1 = x;
        iter = iter + 1;
    end
    error('La méthode de la sécante a échoué pour converger.');
end

// Exemple d'utilisation
f = @(x) x^2 - 2; // Fonction à résoudre
x0 = 1;
x1 = 2;
tol = 1e-6;
max_iter = 1000;
x = secant_method(f, x0, x1, tol, max_iter);
disp(x);
```

5.3.2.4 Méthode de la fausse position

Cette méthode est similaire à la méthode de la dichotomie, mais elle utilise une interpolation linéaire entre les valeurs de la fonction sur les bords de l'intervalle plutôt que de simplement évaluer la fonction au milieu de l'intervalle. Cela peut conduire à une convergence plus rapide, mais il est important de noter que cette méthode peut échouer si la fonction change rapidement entre les bords de l'intervalle.

```
function y = f(x)
    y = x^2 - 2;
endfunction

function x = false_position_method(f, a, b, tol, max_iter)
    if f(a) * f(b) > 0 then
        error('Pas de changement de signe dans l''intervalle
        donné.');
    end

    iter = 0;
    while iter < max_iter
        c = (a * f(b) - b * f(a)) / (f(b) - f(a));
        if f(c) == 0 || abs(f(c)) < tol then
```

```
            x = c;
            return;
        end
        if f(c) * f(a) < 0 then
            b = c;
        else
            a = c;
        end
        iter = iter + 1;
    end
    error('La méthode de la fausse position a échoué pour
    converger.');
endfunction

// Exemple d'utilisation
a = 0;
b = 2;
tol = 1e-6;
max_iter = 1000;
x = false_position_method(f, a, b, tol, max_iter);
disp(x);
```

5.3.2.5 Méthode de la méthode de Brent

Cette méthode combine plusieurs méthodes pour offrir une convergence rapide et une bonne robustesse dans une grande variété de cas. Elle utilise la dichotomie, la sécante et l'interpolation quadratique inverse pour approximer la racine de manière efficace. La méthode de Brent est souvent considérée comme l'une des méthodes les plus fiables pour la résolution d'équations non linéaires.

```
function y = f(x)
    y = x^2 - 2;
endfunction
function x = brent_method(f, a, b, tol, max_iter)
    fa = f(a); // Initialisation des variables
    fb = f(b);
    c = a;
    fc = fa;
    d = 0;
    e = 0;
```

```
        // Vérification du changement de signe dans l'intervalle
    if fa * fb >= 0 then
        error('Pas de changement de signe dans l''intervalle  donné.');
    end
eps_approx = 1e-15; // Valeur approchée de eps
        for iter = 1:max_iter
        // Test de la convergence
        if abs(fb) < abs(fa) then
            a = b;
            b = c;
            c = a;
            fa = fb;
            fb = fc;
            fc = fa;
        end
        tol1 = 2 * eps_approx * abs(b) + 0.5 * tol;
        xm = 0.5 * (c - b);
        if abs(xm) <= tol1 || fb == 0 then
            x = b;
            return;
        end
        // Tenter l'interpolation quadratique inverse
        if abs(e) >= tol1 && abs(fa) > abs(fb) then
            s = fb / fa;
            if a == c then
                p = 2 * xm * s;
                q = 1 - s;
            else
                q = fa / fc;
                r = fb / fc;
                p = s * (2 * xm * q * (q - r) - (b - a)
                * (r - 1));
                q = (q - 1) * (r - 1) * (s - 1);
            end
            if p > 0 then
                q = -q;
            else
                p = -p;
            end
            if 2 * p < 3 * xm * q - abs(tol1 * q) && p < abs
            (0.5 * e * q) then
```

```
                e = d;
                d = p / q;
            else
                d = xm;
                e = d;
            end
        else
            d = xm;
            e = d;
        end
        // Sauvegarde de la dernière valeur
        a = b;
        fa = fb;
        if abs(d) > tol1 then
            b = b + d;
        else
            if xm > 0 then
                b = b + tol1;
            else
                b = b - tol1;
            end
        end
        fb = f(b);
        if (sign(fb) == sign(fc)) then
            c = a;
            fc = fa;
            d = b - a;
            e = d;
        end
    end
    error('La méthode de Brent a échoué pour converger.');
endfunction
// Exemple d'utilisation
a = 0;
b = 2;
tol = 1e-6;
max_iter = 1000;
x = brent_method(f, a, b, tol, max_iter);
disp(x);
```

Travaux Pratique 5 : Calcul numérique

Méthode de la dichotomie

Écrivez une fonction en Scilab qui utilise la méthode de la dichotomie pour trouver une approximation de la racine de l'équation $f(x) = x^2 - 2$ dans l'intervalle $[0, 2]$ avec une tolérance de 10^{-6}.

Méthode de la sécante

Implémentez la méthode de la sécante en Scilab pour résoudre l'équation $f(x) = x^3 - 3x - 5$ avec deux points initiaux $x_0 = 2$ et $x_1 = 3$ et une tolérance de 10^{-5}.

Méthode de la fausse position

Écrivez une fonction en Scilab qui utilise la méthode de la fausse position pour résoudre l'équation $f(x) = \sin(x) - x/2$ dans l'intervalle $[0, 1]$ avec une tolérance de 10^{-4}.

Méthode de Brent

Implémentez la méthode de Brent en Scilab pour trouver une approximation de la racine de l'équation $f(x) = e^{-x} - \cos(x)$ dans l'intervalle $[0, 2]$ avec une tolérance de 10^{-7}.

Résolution de systèmes linéaires

Écrivez une fonction en Scilab qui utilise la méthode de votre choix (Gauss, Jacobi, Gauss-Seidel, relaxation) pour résoudre le système d'équations linéaires suivant :

$$\begin{cases} 2x + y - z = 8 \\ -3x - y + 2z = -11 \\ -2x + y + 2z = -3 \end{cases}$$

Testez votre fonction avec une tolérance de 10^{-6}.

Correction de TP 5 : Calcul numérique

Méthode de la dichotomie

```
function y = f(x)
    y = x^2 - 2;
endfunction

function x = dichotomy_method(f, a, b, tol)
    while (b - a) > tol
        c = (a + b) / 2;
        if f(c) == 0 then
            x = c;
            return;
        elseif f(a) * f(c) < 0 then
            b = c;
        else
            a = c;
        end
    end
    x = (a + b) / 2;
endfunction

// Intervalle initial
a = 0;
b = 2;

// Tolérance
tol = 1e-6;

// Appel de la méthode de la dichotomie
x = dichotomy_method(f, a, b, tol);

// Affichage du résultat
disp(x);
```

Méthode de la sécante

```
function y = f(x)
    y = x^3 - 3*x - 5;
endfunction

function x = secant_method(f, x0, x1, tol)
    while abs(f(x1)) > tol
        x_next = x1 - f(x1) * (x1 - x0) / (f(x1) - f(x0));
        x0 = x1;
        x1 = x_next;
    end
    x = x1;
endfunction

// Points initiaux
x0 = 2;
x1 = 3;

// Tolérance
tol = 1e-5;

// Appel de la méthode de la sécante
x = secant_method(f, x0, x1, tol);

// Affichage du résultat
disp(x);
```

Méthode de la fausse position

```
   // Définition de la fonction f(x) = sin(x) - x/2
function y = f(x)
    y = sin(x) - x/2;
endfunction

// Définition de la fonction de la méthode de la fausse position
function x = false_position_method(f, a, b, tol)
    while abs(b - a) > tol
```

```
        c = (a*f(b) - b*f(a)) / (f(b) - f(a));
        if f(c) == 0 then
            x = c;
            return;
        elseif f(a) * f(c) < 0 then
            b = c;
        else
            a = c;
        end
    end
    x = (a + b) / 2;
endfunction

// Intervalle initial
a = 0;
b = 1;

// Tolérance
tol = 1e-4;

// Appel de la méthode de la fausse position
x = false_position_method(f, a, b, tol);

// Affichage du résultat
disp(x);
```

Méthode de Brent

```
// Désactivation de l'avertissement pour les redéfinitions de
fonctions
    funcprot(0);

function x = brent_method(f, a, b, tol)
    eps_approx = 1e-15; // Valeur approximative de eps
    iter_max = 1000; // Nombre maximal d'itérations
    fa = f(a);
    fb = f(b);
    c = a;
```

```
    fc = fa;
    d = 0;
    e = 0;

    for iter = 1:iter_max
        if (fb > 0 && fc > 0) || (fb < 0 && fc < 0) then
            c = a;
            fc = fa;
            d = b - a;
            e = d;
        end
        if abs(fc) < abs(fb) then
            a = b;
            b = c;
            c = a;
            fa = fb;
            fb = fc;
            fc = fa;
        end
        tol1 = 2 * eps_approx * abs(b) + 0.5 * tol;
        xm = 0.5 * (c - b);
        if abs(xm) <= tol1 || fb == 0 then
            x = b;
            return;
        end
        if abs(e) >= tol1 && abs(fa) > abs(fb) then
            s = fb / fa;
            if a == c then
                p = 2 * xm * s;
                q = 1 - s;
            else
                q = fa / fc;
                r = fb / fc;
                p = s * (2 * xm * q * (q - r) - (b - a) * (r - 1));
                q = (q - 1) * (r - 1) * (s - 1);
            end
            if p > 0 then
                q = -q;
            else
                p = -p;
            end
```

```
            if 2 * p < 3 * xm * q - abs(tol1 * q) && p < abs
            (0.5 * e * q) then
                e = d;
                d = p / q;
            else
                d = xm;
                e = d;
            end
        else
            d = xm;
            e = d;
        end
        a = b;
        fa = fb;
        if abs(d) > tol1 then
            b = b + d;
        else
            if xm > 0 then
                b = b + tol1;
            else
                b = b - tol1;
            end
        end
        fb = f(b);
        if (sign(fb) == sign(fc)) then
            c = a;
            fc = fa;
            d = b - a;
            e = d;
        end
    end
    error('La méthode de Brent n''a pas convergé.');
endfunction

// Réactivation de l'avertissement pour les redéfinitions de fonctions
funcprot(1);

// Définition de la fonction f(x) = exp(-x) - cos(x)
function y = f(x)
    y = exp(-x) - cos(x);
endfunction
```

```

// Intervalle initial
a = 0;
b = 2;

// Tolérance
tol = 1e-7;

// Appel de la méthode de Brent
x = brent_method(f, a, b, tol);

// Affichage du résultat
disp(x);
```

La méthode de Gauss

```
       function x = gauss_elimination(A, b)
    n = size(A, 1);
    Ab = [A, b]; // Matrice augmentée

    // Étape d'élimination
    for k = 1:n-1
        for i = k+1:n
            coef = Ab(i,k) / Ab(k,k);
            for j = k:n+1
                Ab(i,j) = Ab(i,j) - coef * Ab(k,j);
            end
        end
    end

    // Substitution arrière
    x = zeros(n, 1);
    for i = n:-1:1
        x(i) = Ab(i,n+1) / Ab(i,i);
        for j = i+1:n
            x(i) = x(i) - Ab(i,j) * x(j) / Ab(i,i);
        end
    end
```

```
end

// Spécification de la matrice A et du vecteur b
A = [2 1 -1; -3 -1 2; -2 1 2];
b = [8; -11; -3];

// Appel de la fonction gauss_elimination avec une tolérance de 10^-6
x = gauss_elimination(A, b);

// Affichage du résultat
disp(x);
```

La méthode de Jacobi

```
function x = jacobi(A, b, tol, max_iter)
    n = size(A, 1);
    x = zeros(n, 1); // Initialisation de la solution
    x_old = x;

    iter = 0;
    while iter < max_iter
        for i = 1:n
            sigma = 0;
            for j = 1:n
                if j ~= i then
                    sigma = sigma + A(i, j) * x_old(j);
                end
            end
            x(i) = (b(i) - sigma) / A(i, i);
        end
        if norm(x - x_old, 1) < tol then
            return;
        end
        x_old = x;
        iter = iter + 1;
    end
    error('La méthode de Jacobi n''a pas convergé.');
endfunction
```

```
25
26 // Matrice A et vecteur b
27 A = [4 1 1; 1 5 2; 2 4 6];
28 b = [6; 13; 20];
29
30 // Tolérance et nombre maximum d'itérations
31 tol = 1e-6;
32 max_iter = 1000;
33
34 // Appel de la fonction Jacobi
35 x = jacobi(A, b, tol, max_iter);
36
37 // Affichage du résultat
38 disp(x);
```

La méthode de Gauss-Seidel

```
1  function x = gauss_seidel(A, b, tol, max_iter)
2      n = size(A, 1);
3      x = zeros(n, 1); // Initialisation de la solution
4      x_old = x;
5
6      iter = 0;
7      while iter < max_iter
8          for i = 1:n
9              sigma = 0;
10             for j = 1:i-1
11                 sigma = sigma + A(i, j) * x(j);
12             end
13             for j = i+1:n
14                 sigma = sigma + A(i, j) * x_old(j);
15             end
16             x(i) = (b(i) - sigma) / A(i, i);
17         end
18         if norm(x - x_old, 1) < tol then
19             return;
20         end
21         x_old = x;
```

```
        iter = iter + 1;
    end
    error('La méthode de Gauss-Seidel n''a pas convergé.');
endfunction

// Matrice A et vecteur b
A = [4 1 1; 1 5 2; 2 4 6];
b = [6; 13; 20];

// Tolérance et nombre maximum d'itérations
tol = 1e-6;
max_iter = 1000;

// Appel de la fonction Gauss-Seidel
x = gauss_seidel(A, b, tol, max_iter);

// Affichage du résultat
disp(x);
```

La méthode de relaxation

```
function x = relaxation(A, b, omega, tol, max_iter)
    n = size(A, 1);
    x = zeros(n, 1); // Initialisation de la solution

    iter = 0;
    while iter < max_iter
        x_old = x;
        for i = 1:n
            sigma = 0;
            for j = 1:n
                if j ~= i then
                    sigma = sigma + A(i, j) * x(j);
                end
            end
            x(i) = (1 - omega) * x_old(i) + (omega / A(i, i)) * (b(i) -
            ↪  sigma);
        end
```

```
        if norm(x - x_old, 1) < tol then
            return;
        end
        iter = iter + 1;
    end
    error('La méthode de relaxation n''a pas convergé.');
endfunction

// Matrice A et vecteur b
A = [4 1 1; 1 5 2; 2 4 6];
b = [6; 13; 20];

// Paramètres de relaxation, tolérance et nombre maximum d'itérations
omega = 1.2; // Facteur de relaxation
tol = 1e-6;
max_iter = 1000;

// Appel de la fonction de relaxation
x = relaxation(A, b, omega, tol, max_iter);

// Affichage du résultat
disp(x);
```

Chapitre 6

Applications mathématiques

6.1 Les séries et les transformations de Fourier

Les séries et les transformations de Fourier représentent des outils mathématiques puissants qui trouvent une utilité étendue dans divers domaines, notamment les mathématiques, la physique, l'ingénierie et les sciences de la vie. Ce chapitre a pour objectif d'explorer ces concepts avancés et de mettre en lumière leur application pratique au moyen de Scilab, un environnement de programmation scientifique. Vous aborderez les fondements théoriques des séries et transformations de Fourier, mettrez en évidence leurs implications dans la résolution de problèmes concrets, et démontrerez comment les implémenter efficacement à l'aide des fonctionnalités disponibles dans Scilab. En combinant la théorie et la pratique, nous visons à fournir aux lecteurs une compréhension approfondie de ces outils mathématiques essentiels et à les habiliter à les appliquer avec succès dans leurs propres domaines d'étude ou de travail.

6.1.1 Les séries de Fourier

Les séries de Fourier constituent une méthode puissante pour représenter une fonction périodique sous la forme d'une somme infinie de fonctions sinusoïdales. Cette approche repose sur la décomposition d'une fonction périodique en une série de termes harmoniques, chacun caractérisé par une fréquence, une amplitude et une phase spécifiques.

Scilab facilite le calcul des coefficients de la série de Fourier d'une fonction périodique donnée. Les fonctions intégrées telles que **fft** (transformée de Fourier discrète) et **ifft** (transformée de Fourier inverse) dans Scilab sont des outils essentiels pour effectuer ces calculs de manière efficace. De plus, l'utilisation de la fonction plot permet de visualiser aisément le spectre de fréquence associé à une fonction périodique.

Ainsi, grâce à Scilab, vous avez à votre disposition des outils performants pour explorer et analyser les composantes harmoniques d'une fonction pério-

dique, ouvrant ainsi la voie à une compréhension approfondie des caractéristiques fréquentielles de ces fonctions.

6.1.2 Les transformations de Fourier

Les transformations de Fourier représentent une extension des séries de Fourier adaptée aux fonctions non périodiques. Elles permettent de représenter une fonction dans le domaine des fréquences en recourant à une transformée mathématique. Ces transformations trouvent une large application dans des domaines tels que le traitement du signal, l'imagerie médicale, la compression de données, et diverses autres applications.

Scilab propose un ensemble de fonctions, dont **fft** et ifft, destinées au calcul des transformations de Fourier d'une fonction donnée. La fonction **fftshift** peut également être exploitée pour recentrer le spectre de fréquence, améliorant ainsi la visualisation des résultats. De plus, Scilab met à disposition des outils efficaces pour le calcul de la transformée de Fourier rapide **(FFT)** et de la transformée de Fourier discrète inverse **(IFFT)**, offrant ainsi des solutions performantes pour les analyses fréquentielles dans divers contextes.

Grâce à ces fonctionnalités, Scilab s'avère être un environnement robuste pour explorer et comprendre les caractéristiques fréquentielles des fonctions non périodiques, ouvrant ainsi la voie à des applications variées et innovantes dans le domaine de l'analyse et du traitement du signal.

6.1.3 Applications des séries et des transformations de Fourier

Les séries et les transformations de Fourier sont des outils mathématiques aux applications variées et pratiques. Voici quelques exemples concrets illustrant leur utilité :

— Traitement du signal : Les séries et les transformations de Fourier jouent un rôle essentiel dans l'analyse et le filtrage des signaux. Leur utilisation est répandue dans des secteurs tels que la télécommunication, le traitement audio et vidéo, permettant ainsi la décomposition et la manipulation efficace des signaux.

— Imagerie médicale : Les transformations de Fourier sont cruciales dans la reconstruction d'images à partir de données médicales, notamment dans les domaines de la tomographie par ordinateur (CT scan), de l'imagerie par résonance magnétique (IRM) et d'autres techniques d'imagerie médicale. Elles contribuent à obtenir des images détaillées et précises pour le diagnostic et le suivi médical.

— Compression de données : Les transformations de Fourier sont intégrées aux algorithmes de compression de données, tels que JPEG. Elles permettent de réduire la taille des fichiers tout en préservant la qualité de l'image. Cette application est essentielle dans le stockage et le transfert efficaces d'informations visuelles.

— Analyse de fréquence : Les séries et les transformations de Fourier sont inestimables pour analyser les composantes fréquentielles présentes dans un signal. Cette capacité d'analyse est largement exploitée dans des domaines tels que l'analyse des vibrations, l'acoustique et la musique, où la compréhension précise des fréquences est cruciale.

Ainsi, ces outils mathématiques avancés jouent un rôle central dans diverses applications, contribuant significativement aux avancées technologiques et scientifiques dans des domaines aussi variés que le traitement du signal, l'imagerie médicale, la compression de données, et l'analyse de fréquence.

6.1.4 Exemple d'utilisation dans Scilab

Pour illustrer l'utilisation des séries et des transformations de Fourier dans Scilab, considérons l'exemple suivant : Supposons que vous ayez une fonction périodique **f(t)** avec une période T. Vous pouvez calculer les coefficients de la série de Fourier de **f(t)** à l'aide de la fonction **fft** de Scilab. Ensuite, vous pouvez tracer le spectre de fréquence de **f(t)** à l'aide de la fonction plot.

```
// Définition de la fonction périodique f(t)
function y = f(t)
    y = sin(2*%pi*t) + 0.5*sin(4*%pi*t) + 0.25*sin(6*%pi*t);
endfunction

// Calcul des coefficients de la série de Fourier
N = 100; // Nombre de termes de la série
T = 1; // Période de la fonction
t = linspace(0, T, N);
coefficients = fft(f(t));

// Tracé du spectre de fréquence
frequencies = linspace(0, 1/T, N);
frequencies = frequencies(1:N/2);
// Utilisez uniquement la moitié des fréquences (partie positive)
coefficients = coefficients(1:N/2);
// Utilisez uniquement la moitié des coefficients (partie positive)

plot(frequencies, abs(coefficients));
xtitle('Spectre de fréquence', 'Fréquence (Hz)', 'Amplitude');
```

Remarque

Ce code calcule les coefficients de la série de Fourier de la fonction périodique f(t) et trace le spectre de fréquence correspondant. Vous pouvez expérimenter avec différentes fonctions périodiques et périodes pour explorer davantage les séries et les transformations de Fourier dans Scilab.

En conclusion, les séries et les transformations de Fourier s'avèrent être des outils indispensables dans l'analyse et le traitement de signaux et de fonctions. Leur utilité s'étend à des domaines variés tels que le traitement du signal, l'imagerie médicale, la compression de données, et l'analyse de fréquence. Scilab, grâce à ses fonctionnalités puissantes, offre un environnement propice au calcul et à la visualisation de ces concepts mathématiques avancés. Cela permet aux chercheurs, ingénieurs et scientifiques d'explorer et d'exploiter pleinement les avantages de ces techniques pour résoudre des problèmes complexes et faire progresser les connaissances dans divers domaines.

6.2 Les équations aux dérivées partielles

La résolution d'équations aux dérivées partielles (EDP) dépend fortement du type spécifique d'équations que vous avez et des conditions aux limites associées. Scilab offre des outils numériques puissants qui peuvent être utilisés pour résoudre numériquement des EDP. Voici un exemple de résolution d'une équation de la chaleur (équation de la diffusion) 1D en utilisant la méthode des différences finies avec Scilab :

6.2.1 La méthode des différences finies

Dans cet exemple, vous utiliserez la méthode des différences finies pour résoudre l'équation de la chaleur unidimensionnelle. La solution est obtenue de manière itérative en discrétisant l'espace et le temps.

Si vous avez une équation aux dérivées partielles spécifique en tête ou si vous avez des conditions aux limites particulières, n'hésitez pas à fournir plus de détails pour obtenir une aide plus ciblée.

```
// Paramètres physiques
alpha = 0.01; // coefficient de diffusion
L = 1.0; // longueur du domaine
T = 1.0; // temps final

// Discrétisation du domaine en espace et en temps
Nx = 50; // nombre de points en espace
Nt = 100; // nombre de pas de temps
```

```

dx = L / (Nx - 1);
dt = T / Nt;

// Conditions initiales et aux limites
u0 = zeros(Nx, 1); // condition initiale
u0(1) = 1.0; // condition aux limites à gauche

u = u0;
// Boucle méthode des différences finies explicites
for n = 1:Nt
    u_new = u;

    for i = 2:Nx-1
      u_new(i) =
      u(i) + alpha * dt / (dx^2) * (u(i+1) - 2*u(i) + u(i-1));
    end

    u = u_new;
end

// Tracé des résultats
x = linspace(0, L, Nx)';
plot(x, u, '-o');
xlabel('Position');
ylabel('Température');
title('Équation de la chaleur 1D');
```

6.2.2 La Méthode des éléments finis

La méthode des éléments finis (MEF) est une technique numérique utilisée pour résoudre des équations aux dérivées partielles (EDP) en discrétisant le domaine en éléments finis. Chaque élément est approximé par une fonction de forme, et la solution globale est obtenue en assemblant ces éléments. Voici un exemple simple de résolution de l'équation de Laplace 1D avec la méthode des éléments finis en utilisant Scilab :

```
// Paramètres physiques
L = 1.0; // longueur du domaine
Ne = 10; // nombre d'éléments
```

```
Nn = Ne + 1; // nombre de nœuds

x_nodes = linspace(0, L, Nn)';// Discrétisation du domaine

// Définition des éléments
elements = zeros(Ne, 2);
for i = 1:Ne
    elements(i,:) = [i, i+1];
end

// Assemblage de la matrice de rigidité
K = zeros(Nn, Nn);
for e = 1:Ne
    nodes = elements(e,:);
    x_e = x_nodes(nodes);
    h = x_e(2) - x_e(1);
    ke = 1 / h * [1, -1; -1, 1];
    K(nodes, nodes) = K(nodes, nodes) + ke;
end

// Conditions aux limites
K(1,:) = 0;
K(:, 1) = 0;
K(1, 1) = 1;
K(Nn,:) = 0;
K(:, Nn) = 0;
K(Nn, Nn) = 1;

f = zeros(Nn, 1);// Vecteur de charge
// Charge à la fin du domaine
f(Nn) = 1.0;
// Résolution du système linéaire
u = K \ f;

// Tracé des résultats
plot(x_nodes, u, '-o');
xlabel('Position');
ylabel('Déplacement');
title('Méthode des éléments finis - Équation de Laplace 1D');
```

6.2.3 La Méthode des volumes finis

La méthode des volumes finis est une technique numérique utilisée pour résoudre des équations aux dérivées partielles (EDP) en discrétisant le domaine en volumes finis et en approximant les intégrales sur chaque volume. Voici un exemple simple de résolution de l'équation de la chaleur 1D avec la méthode des volumes finis en utilisant Scilab :

```
// Paramètres physiques
L = 1.0; // longueur du domaine
Nx = 10; // nombre de nœuds
dx = L / (Nx - 1); // pas spatial

x_nodes = linspace(0, L, Nx)';// Discrétisation du domaine
// Conditions initiales et aux limites
u0 = zeros(Nx, 1); // condition initiale
u0(1) = 1.0; // condition aux limites à gauche

u = u0;

// Paramètres de la simulation
dt = 0.001; // pas de temps
T = 0.1; // temps final
Nt = round(T / dt); // nombre de pas de temps

// Boucle temporelle (méthode des volumes finis)
for n = 1:Nt
    u_new = u;

    for i = 2:Nx-1
        flux = (u(i) - u(i-1)) / dx; // flux numérique
        u_new(i) = u(i) - dt / dx * flux;
    end

    u = u_new;
end
// Tracé des résultats
plot(x_nodes, u, '-o');
xlabel('Position');
ylabel('Température');
title('Méthode des volumes finis - Équation de la chaleur 1D');
```

6.3 Les méthodes numériques avancées

Les méthodes numériques avancées constituent un ensemble de techniques mathématiques déployées pour résoudre des problèmes complexes qui échappent aux solutions analytiques. Elles se révèlent particulièrement indispensables dans les domaines des sciences et de l'ingénierie, où les modélisations et simulations de phénomènes réels requièrent fréquemment des calculs mathématiques complexes.

Ce chapitre se penchera sur diverses méthodes numériques avancées intégrées dans Scilab, offrant ainsi des solutions pour une gamme étendue de problèmes mathématiques. Ces méthodes reposent sur des algorithmes numériques sophistiqués qui garantissent la précision et la fiabilité des résultats obtenus.

6.3.1 Méthode de Newton-Raphson

La méthode de Newton-Raphson est une technique numérique utilisée pour trouver les racines d'une équation. Elle nécessite une approximation initiale de la racine et converge rapidement vers une solution plus précise. Voici un exemple d'implémentation de la méthode de Newton-Raphson en utilisant Scilab :

```
// Définir la fonction pour laquelle nous recherchons la racine
function y = f(x)
    y = x^2 - 4;
endfunction

// Définir la dérivée de la fonction
function y = df(x)
    y = 2*x;
endfunction

// Implémentation de la méthode de Newton-Raphson
function root = newtonRaphson(initial_guess, tolerance, max_iterations)
    x = initial_guess;
    iter = 0;

    while abs(f(x)) > tolerance && iter < max_iterations
        x = x - f(x) / df(x);
        iter = iter + 1;
    end

    root = x;
endfunction

```

```
// Paramètres
initial_guess = 2.0; // Approximation initiale de la racine
tolerance = 1e-6;   // Tolérance pour arrêter l'itération
max_iterations = 100; // Nombre maximal d'itérations

// Appel de la méthode de Newton-Raphson
result = newtonRaphson(initial_guess, tolerance, max_iterations);

// Affichage du résultat
disp('Racine trouvée:');
disp(result);
```

6.3.2 Méthode des différences finies

La méthode des différences finies est couramment utilisée pour résoudre des équations aux dérivées partielles (EDP) en discrétisant l'espace et le temps. Voici un exemple simple d'application de la méthode des différences finies pour résoudre l'équation de la chaleur unidimensionnelle avec Scilab :

```
// Paramètres physiques
L = 1.0; // Longueur du domaine
Nx = 50; // Nombre de points en espace
dx = L / (Nx - 1); // Pas spatial

// Conditions initiales et aux limites
u0 = zeros(Nx, 1); // Condition initiale
u0(1) = 100.0; // Condition aux limites à gauche

u = u0;

// Paramètres de la simulation
dt = 0.01; // Pas de temps
T = 1.0; // Temps final
Nt = round(T / dt); // Nombre de pas de temps

// Coefficient de diffusion thermique
alpha = 0.01;

// Boucle temporelle (méthode des différences finies)
for n = 1:Nt
```

```
    u_new = u;

    for i = 2:Nx-1
        u_new(i) = u(i) + alpha * dt / dx^2 * (u(i+1) - 2*u(i) +
        ↪  u(i-1));
    end

    u = u_new;
end

// Tracé des résultats
x = linspace(0, L, Nx)';
plot(x, u, '-o');
xlabel('Position');
ylabel('Température');
title('Méthode des différences finies - Équation de la chaleur');
```

6.3.3 Méthode de Monte Carlo

La méthode de Monte Carlo est une technique numérique basée sur le principe de simulation aléatoire pour résoudre des problèmes mathématiques. Voici un exemple simple d'application de la méthode de Monte Carlo pour estimer la valeur de π en utilisant Scilab :

```
// Fonction pour estimer pi avec la méthode de Monte Carlo
function estimatePi(numsamples)
    inside_circle = 0;

    for i = 1:numsamples
        x = rand();
        y = rand();

        distance = sqrt(x^2 + y^2);

        if distance <= 1.0
            inside_circle = inside_circle + 1;
        end
    end

    pi_estimate = 4 * inside_circle / numsamples;
```

```
    disp('Estimation de pi avec la méthode de Monte Carlo:');
    disp(pi_estimate);
endfunction

// Nombre d'échantillons
num_samples = 100000;

// Appel de la fonction pour estimer pi
estimatePi(num_samples);
```

Remarque

Dans cet exemple, la fonction **estimatePi** génère des points aléatoires dans un carré unité et compte combien d'entre eux se situent à l'intérieur du cercle unité. La proportion de points à l'intérieur du cercle est utilisée pour estimer la valeur de π.
Vous pouvez ajuster le nombre d'échantillons **(numsamples)** pour obtenir une estimation plus précise de π. Notez que l'exactitude de l'estimation dépend du nombre d'échantillons générés.
N'hésitez pas à personnaliser ce code en fonction de vos besoins spécifiques ou si vous avez des questions supplémentaires.

6.3.4 Méthode des éléments finis

La méthode des éléments finis (MEF) est une technique numérique utilisée pour résoudre des équations aux dérivées partielles (EDP) en discrétisant le domaine en éléments finis. Chaque élément est approximé par une fonction de forme, et la solution globale est obtenue en assemblant ces éléments. Voici un exemple simple d'application de la méthode des éléments finis pour résoudre l'équation de Laplace en 1D avec Scilab :

```
// Paramètres physiques
L = 1.0; // Longueur du domaine
Ne = 10; // Nombre d'éléments
Nn = Ne + 1; // Nombre de nœuds

// Discrétisation du domaine
x_nodes = linspace(0, L, Nn)';

// Définition des éléments
elements = zeros(Ne, 2);
```

```
for i = 1:Ne
    elements(i,:) = [i, i+1];
end

// Assemblage de la matrice de rigidité
K = zeros(Nn, Nn);
for e = 1:Ne
    nodes = elements(e,:);
    x_e = x_nodes(nodes);
    h = x_e(2) - x_e(1);

    ke = 1 / h * [1, -1; -1, 1];

    K(nodes, nodes) = K(nodes, nodes) + ke;
end

// Conditions aux limites
K(1,:) = 0;
K(:, 1) = 0;
K(1, 1) = 1;

K(Nn,:) = 0;
K(:, Nn) = 0;
K(Nn, Nn) = 1;

// Vecteur de charge
f = zeros(Nn, 1);
f(Nn) = 1.0; // Charge à la fin du domaine

// Résolution du système linéaire
u = K \ f;

// Tracé des résultats
plot(x_nodes, u, '-o');
xlabel('Position');
ylabel('Déplacement');
title('Méthode des éléments finis - Équation de Laplace 1D');
```

Travaux Pratique 6 : Applications mathématiques

Séries de Fourier

1. Écrivez une fonction Scilab pour calculer les coefficients de Fourier d'une fonction périodique donnée.
2. Utilisez la fonction précédente pour approximer une fonction périodique donnée par sa série de Fourier tronquée à différents ordres.
3. Tracez la fonction originale et ses approximations en utilisant différentes valeurs d'ordre de la série de Fourier.

Transformée de Fourier

1. Écrivez une fonction Scilab pour calculer la transformée de Fourier discrète (DFT) d'un signal donné.
2. Utilisez la DFT pour analyser les composantes fréquentielles d'un signal quelconque.
3. Comparez la transformée de Fourier discrète avec la transformée de Fourier continue pour une fonction périodique.

Équations de la chaleur et de l'onde

1. Implémentez une méthode numérique (par exemple la méthode des différences finies) pour résoudre une équation de la chaleur unidimensionnelle.
2. Utilisez votre implémention pour simuler la diffusion de la chaleur dans une barre métallique avec des conditions initiales et aux limites spécifiées.
3. Modifiez votre implémentation pour résoudre une équation d'onde unidimensionnelle et visualisez l'évolution de la vague à différents moments dans le temps.

Méthode de Décomposition de Domaine

1. Implémentez une méthode de décomposition de domaine pour résoudre une équation aux dérivées partielles (par exemple, une équation de la chaleur ou une équation d'onde) sur un domaine bidimensionnel.
2. Divisez le domaine en plusieurs sous-domaines et résolvez l'équation sur chaque sous-domaine de manière indépendante.
3. Utilisez des conditions aux limites appropriées pour assurer la continuité de la solution sur les interfaces entre les sous-domaines.
4. Comparez les performances de la méthode de décomposition de domaine avec une méthode de résolution directe pour des problèmes de taille similaire.

Correction de TP 6 : Applications mathématiques

Séries de Fourier

```
// Fonction pour calculer les coefficients de Fourier d'une fonction périodique
function [a0, an, bn] = coefficients_fourier(f, T, N)
    // Calcul du coefficient a0
    a0 = (1 / T) * integrate_trapezoidal(f, 0, T, 1000);

    // Initialisation des vecteurs pour stocker les coefficients a_n et b_n
    an = zeros(1, N);
    bn = zeros(1, N);

    // Calcul des coefficients a_n et b_n
    for n = 1:N
        // Définir les fonctions à intégrer
        function f_cos_t = f_cos(t)
            f_cos_t = f(t) * cos(2 * %pi * n * t / T);
        endfunction

        function f_sin_t = f_sin(t)
            f_sin_t = f(t) * sin(2 * %pi * n * t / T);
        endfunction

        // Intégrer les fonctions avec la méthode des trapèzes
        an(n) = (2 / T) * integrate_trapezoidal(f_cos, 0, T, 1000);
        bn(n) = (2 / T) * integrate_trapezoidal(f_sin, 0, T, 1000);
    end
endfunction

// Définir la fonction pour l'intégration numérique avec la méthode des trapèzes
function result = integrate_trapezoidal(f, a, b, n)
    h = (b - a) / n;
    result = 0;
    result = result + f(a) / 2;
    for i = 1:(n-1)
        result = result + f(a + i * h);
```

```
    end
    result = result + f(b) / 2;
    result = result * h;
endfunction

// Définir la fonction périodique à approximer
function y = f(t)
    y = sin(t); // Exemple de fonction, remplacez par votre propre fonction
endfunction
// Période de la fonction
T = 2 * %pi;

// Nombre d'ordres pour la série de Fourier
N = 5;
// Calcul des coefficients de Fourier
[a0, an, bn] = coefficients_fourier(f, T, N);
// Fonction pour approximer la fonction périodique par sa série de Fourier
function y = approximation_fourier(t, a0, an, bn, T, N)
    y = a0 / 2;
    for n = 1:N
        y = y + an(n) * cos(2 * %pi * n * t / T) + bn(n) * sin(2 * %pi * n * t / T);
    end
endfunction

// Tracé de la fonction originale et de ses approximations
t = linspace(0, T, 1000);
y_original = f(t);
y_approximation = approximation_fourier(t, a0, an, bn, T, N);

clf;
plot(t, y_original, 'b', t, y_approximation, 'r');
legend('Fonction originale', 'Approximation de Fourier');
xlabel('t');
ylabel('f(t)');
title('Approximation de Fourier d une fonction périodique');
```

Transformée de Fourier

```
// Fonction pour calculer la transformée de Fourier discrète (DFT)
function X = dft(x)
    N = length(x);
    X = zeros(1, N);
    for k = 0:N-1
        X(k+1) = sum(x .* exp(-%i*2*%pi*k*(0:N-1)/N));
    end
endfunction

// Génération d'un signal quelconque
t = 0:0.01:2*%pi;
x = sin(2*%pi*5*t) + 0.5*sin(2*%pi*10*t);

// Calcul de la DFT du signal
X = dft(x);

// Affichage du résultat
plot(abs(X))
xlabel('Fréquence')
ylabel('Amplitude')
title('Transformée de Fourier discrète du signal')

```

Équations de la chaleur et de l'onde

```
// Paramètres physiques
L = 1.0;  // Longueur de la barre
T = 1.0;  // Temps final de la simulation
Nx = 100;  // Nombre de points sur la grille spatiale
Nt = 1000;  // Nombre de points sur la grille temporelle
alpha = 0.01;  // Coefficient de diffusion thermique

// Discrétisation spatiale et temporelle
dx = L / Nx;  // Taille de l'intervalle spatial
dt = T / Nt;  // Taille de l'intervalle temporel

```

```
// Conditions initiales
u0 = zeros(1, Nx);  // Température initiale
u0(int(0.4 * Nx):int(0.6 * Nx)) = 1.0;  // Condition initiale non nulle

// Fonction pour résoudre l'équation de la chaleur par la méthode des différences finies
function u_final = solve_heat_equation(u0, alpha, dx, dt, Nt)
    u = u0;
    for t = 1:Nt
        u(2:$-1) = u(2:$-1) + alpha * dt / dx^2 * (u(1:$-2) - 2 * u(2:$-1) + u(3:$));
    end
    u_final = u;
endfunction

// Résolution de l'équation de la chaleur
u_final = solve_heat_equation(u0, alpha, dx, dt, Nt);

// Visualisation
x = linspace(0, L, Nx);
plot(x, u_final);
xlabel('Position');
ylabel('Température');
title('Diffusion de la chaleur dans une barre métallique');
grid on;
```

Méthode de Décomposition de Domaine

```
// Définition de la taille du domaine et du nombre de sous-domaines
Nx = 100; // Nombre de points dans la direction x
Ny = 100; // Nombre de points dans la direction y
num_subdomains_x = 4; // Nombre de sous-domaines dans la direction x
num_subdomains_y = 4; // Nombre de sous-domaines dans la direction y

// Définition de la discrétisation du domaine
dx = 1.0 / Nx; // Pas de discrétisation dans la direction x
dy = 1.0 / Ny; // Pas de discrétisation dans la direction y

```

```
// Initialisation de la solution
u = zeros(Nx, Ny);

// Boucle sur les sous-domaines
for i = 1:num_subdomains_x
    for j = 1:num_subdomains_y
        // Définition des limites du sous-domaine
        start_x = (i - 1) * Nx / num_subdomains_x + 1;
        end_x = i * Nx / num_subdomains_x;
        start_y = (j - 1) * Ny / num_subdomains_y + 1;
        end_y = j * Ny / num_subdomains_y;

        // Résolution de l'équation sur le sous-domaine (exemple simple)
        // Ici, nous initialisons simplement le sous-domaine avec une valeur constante pour illustrer
        // La résolution de l'équation pourrait nécessiter l'utilisation d'autres méthodes numériques telles que la méthode des différences finies, la méthode des éléments finis, etc.
        u(start_x:end_x, start_y:end_y) = 1.0;
    end
end

// Traitement des conditions aux limites pour garantir la continuité de la solution sur les interfaces entre les sous-domaines
// (À implémenter en fonction des conditions spécifiques de votre équation)

// Comparaison des performances avec une méthode de résolution directe
// (À implémenter en fonction de la méthode de résolution directe utilisée)
```

Conclusion

En conclusion, ce polycopié a été soigneusement élaboré pour fournir aux étudiants les bases essentielles nécessaires à la maîtrise des outils de programmation, en mettant l'accent sur le langage scientifique Scilab. À travers ses six chapitres, nous avons abordé progressivement les différentes facettes de Scilab, tout en intégrant des travaux pratiques pour une application concrète des connaissances acquises.

- **Chapitre 1** : Nous avons introduit Scilab en détaillant les étapes d'installation, en présentant l'interface de développement intégré (IDE), l'éditeur, la manière d'exécuter les scripts, ainsi que la fenêtre graphique. De plus, nous avons couvert l'installation des boîtes à outils, avec des instructions détaillées sur deux méthodes possibles. Ce chapitre se conclut par un travail pratique visant à familiariser les étudiants avec l'environnement Scilab.
- **Chapitre 2** : S'est concentré sur les bases de Scilab, notamment les variables et les types de données, les opérations mathématiques, les fonctions et scripts, ainsi que les structures de contrôle. Chaque sous-section est accompagnée d'un travail pratique pour renforcer la compréhension des concepts enseignés.
- **Chapitre 3** : A approfondi la manipulation de données, en couvrant les tableaux, matrices et vecteurs, avec des sections détaillées sur la création, l'accès aux éléments, les opérations et la manipulation avancée. Des exercices pratiques sont également inclus pour permettre aux étudiants de mettre en pratique leurs compétences en manipulation de données.
- **Chapitre 4** : Nous avons exploré la création de graphiques et la visualisation des données avec Scilab. Les différents types de graphiques, la personnalisation, les animations et les diagrammes sont détaillés, avec des exemples pratiques pour illustrer chaque concept.
- **Chapitre 5** : A abordé les calculs numériques, notamment les méthodes numériques, les intégrations numériques et les résolutions d'équations. Chaque sous-section est accompagnée d'exemples pratiques pour aider les étudiants à comprendre et à appliquer les techniques numériques.

— **Chapitre 6** : Traité des applications mathématiques avancées telles que les séries et transformations de Fourier, les équations aux dérivées partielles et les méthodes numériques avancées. Des exemples d'utilisation dans Scilab sont fournis pour chaque sujet abordé.

En somme, ce polycopié a été conçu dans le but de guider les étudiants à travers un parcours d'apprentissage progressif et structuré, en combinant théorie et pratique pour une compréhension approfondie de Scilab. Nous sommes convaincus que la maîtrise de Scilab ouvrira de nombreuses portes dans le domaine de la programmation scientifique, et nous espérons sincèrement que ce polycopié vous a été d'une grande utilité dans cette entreprise.

Enfin, nous vous adressons nos meilleurs vœux de réussite dans vos études et projets à venir, et nous vous encourageons à continuer d'explorer et de perfectionner vos compétences en programmation avec Scilab. N'hésitez pas à nous solliciter pour toute question ou assistance supplémentaire que vous pourriez nécessiter dans votre apprentissage.

Bibliographie

1. *Scilab : A Practical Introduction to Programming and Problem-Solving* par Tejas Sheth, Wiley, 2016.

2. *Scilab by Example* par Étienne Grossmann, Springer, 2015.

3. *Scilab : User Guide* par l'équipe Scilab. Disponible en ligne sur le site officiel de Scilab.

4. *Modeling and Simulation in Scilab/Scicos with ScicosLab 4.4* par Stephen L. Campbell, Jean-Philippe Chancelier, and Ramine Nikoukhah, Springer, 2006.

5. *Scilab from Theory to Practice - I. Fundamentals* par Claude Gomez et al., Springer, 2015.

6. *Scilab - De la théorie à la pratique - Tome 2, Simulation numérique avec Scilab* par Dominique Guegan, Springer, 2007.

7. *Scilab : Advanced Computational Methods and Applications* par Luca Zamboni, CRC Press, 2020.

8. *Scilab : An Open Source Alternative for Scientific Computing* par Satish Annigeri, Apress, 2018.

9. *Scilab : Hands-On Workshop* par Gourab Sen Gupta, BPB Publications, 2019.

10. Tejas Sheth. *Scilab : A Practical Introduction to Programming and Problem Solving*. Wiley, 2016.

11. Étienne Grossmann. *Scilab by Example*. Springer, 2015.

12. l'équipe Scilab. *Scilab : User Guide*. Disponible en ligne sur le site officiel de Scilab.

13. Stephen L. Campbell, Jean-Philippe Chancelier, and Ramine Nikoukhah. *Modeling and Simulation in Scilab/Scicos with ScicosLab 4.4*. Springer, 2006.

14. Claude Gomez et al. *Scilab from Theory to Practice - I. Fundamentals*. Springer, 2015.

15. Dominique Guegan. *Scilab - De la théorie à la pratique - Tome 2, Simulation numérique avec Scilab*. Springer, 2007.

16. Satish Annigeri. *Scilab : An Open Source Alternative for Scientific Computing*. Apress, 2018.

17. Gourab Sen Gupta. *Scilab : Hands-On Workshop*. BPB Publications, 2019.