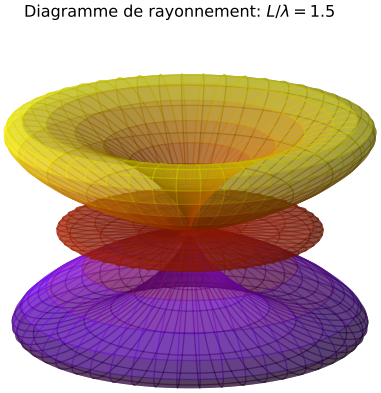
République Tunisienne
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université de Carthage

Institut Préparatoire aux Études Scientifiques et Techniques

Cours & TD de Physique Numérique (Python)

Agrégation en Sciences Mathématiques et Physiques



Ahmed Ammar
Enseignant Contractuel

2019-2022

Table des matières

1 Introduction à Python I : Présentation & installation **5**

1.1 Objectifs généraux en premier 5

1.2 Situation standard que nous rencontrons quotidiennement 5

1.3 Langage Python 6

1.4 Installation d'un environnement Python scientifique 7

1.4.1 Installation sur ordinateur 7

1.4.2 Spyder 8

1.4.3 Installation sur smartphone 8

2 Introduction à Python II : syntaxe et variables **11**

2.1 Introduction : "Hello World !" 11

2.2 Commentaires 11

2.3 Nombres 12

2.4 Affectations (ou assignation) 13

2.4.1 variables 13

2.4.2 Noms de variables réservés (keywords) 14

2.4.3 Les types 14

2.5 Lectures complémentaires 20

2.6 Travaux dirigés 21

2.6.1 Exercice 1 : Les variables 21

2.6.2 Exercice 2 : Fonction `input()` 21

2.6.3 Exercice 3 : Corriger l'erreur dans le code 21

3 Introduction à Python III : Contrôle du flux d'instructions **23**
3.1 Les conditions . . . 23
3.1.1 L'instruction `if` . . . 23
3.1.2 L'instruction `else` . . . 24
3.1.3 L'instruction `elif` . . . 25
3.1.4 Exercice : Condition sur le jour de travail . . . 26
3.2 Les boucles . . . 26
3.2.1 L'instruction `while` . . . 26
3.2.2 L'instruction `for` . . . 27
3.2.3 Exercice : produit de Wallis . . . 28
3.2.4 Compréhensions de listes . . . 29
3.2.5 L'instruction break . . . 30
3.3 Les fonctions . . . 30
3.3.1 Intérêt des fonctions . . . 31
3.3.2 L'instruction def . . . 31
3.4 Les Scripts . . . 32
3.5 Lectures complémentaires . . . 33
3.6 Travaux dirigés . . . 33
3.6.1 Exercice 1 : Comparer deux entiers . . . 33
3.6.2 Exercice 2 : Comparer deux chaînes . . . 33
3.6.3 Exercice 3 : Convertir Euro contre Dinar Tunisien | EUR TND . . . 33
3.6.4 Exercice 4 : Résolution d'une équation du second degré . . . 33
3.6.5 Exercice 5 : programmez une boucle `while` . . . 35
3.6.6 Exercice 6 : Créer une liste avec une boucle `while` . . . 35
3.6.7 Exercice 7 : Programmer une boucle `for` . . . 36
3.6.8 Exercice 8 : Écrire une fonction Python . . . 36
3.6.9 Exercice 9 : Renvoyer trois valeurs d'une fonction Python . . . 36

4 Bibliothèques scientifiques : numpy et matplotlib **37**
4.1 Bibliothèque numérique : `numpy` . . . 37
4.1.1 Tableaux et matrices . . . 37
4.1.2 Lecture et écriture de données . . . 41
4.2 Bibliothèque Python de visualisation des données : `matplotlib` . . . 43
4.2.1 Documentation en ligne et Galerie . . . 43

4.2.2 Guide de Démarrage 44
4.2.3 Vues en grille 45
4.2.4 Commandes de texte de base 45
4.2.5 Styles de lignes et de marqueurs 47
4.2.6 Colormap : Tracés contour, Imshow et 3D 48
4.3 Travaux dirigés 50
4.3.1 Exercice 1 : Tracer une fonction 50
4.3.2 Exercice 2 : Tracer deux fonctions 51
4.3.3 Exercice 3 : Racines d'une équation du second degré 51
4.3.4 Exercice 4 : Approximer une fonction par une somme de sinus 52
4.3.5 Exercice 5 : Fonctions spéciales (intégrales de Fresnel et spirale de Cornu) 53
4.4 TP : Diagramme de rayonnement 56
4.4.1 Simulation Python 56
4.4.2 Antenne dipolaire 58
4.4.3 Simulation Python 59

5 Intégration numérique 62
5.1 Introduction 62
5.2 Idées de base de l'intégration numérique 62
5.2.1 Exemple de calcul 63
5.3 La règle du trapèze composite 63
5.3.1 La formule générale 65
5.3.2 Implémentation 66
5.4 La méthode du point milieu composite 68
5.4.1 L'idée 68
5.4.2 La formule générale 69
5.4.3 Implémentation 69
5.4.4 Comparaison des méthodes du trapèze et du point milieu 70
5.5 Intégration Monte Carlo 71
5.5.1 Exemple : détermination de π 71
5.5.2 implémentation 72
5.6 Travaux dirigés 73
5.6.1 Exercice 1 : Vitesse d'une fusée 73
5.6.2 Exercice 2 : Valeur approchée de π 74

5.6.3 Exercice 3 : Intégration adaptative 74
5.6.4 Exercice 4 : Intégration de x élevé à x 74
5.6.5 Exercice 5 : Orbitales atomiques 75

6 Équations différentielles ordinaires 77
6.1 Introduction 77
6.2 Exemple I : Radioactivité 77
6.2.1 La découverte de la radioactivité 77
6.2.2 Loi de désintégration radioactive 78
6.3 Mouvement d'un projectile 81
6.4 Convergence et de stabilité de la méthode d'Euler : Cas des systèmes linéaires 85
6.4.1 La méthode d'Euler explicite (progressive) 86
6.4.2 La méthode d'Euler implicite (rétrograde) 87
6.4.3 Exemple : Oscillateur libre amorti [masse, ressort, amortisseur] 88
6.4.4 Conclusion 92
6.5 La méthode de Runge-Kutta d'ordre 4 94
6.5.1 Algorithme de Runge-Kutta d'ordre 4 94
6.5.2 Exemple : Système dynamique différentiel de Lorenz (attracteur de Lorenz) 94
6.6 Travaux dirigés 97
6.6.1 Exercice 1 : Pendule simple 97
6.6.2 Exercice 2 : Comparaison des schémas d'Euler explicite et implicite . 100
6.6.3 Exercice 3 : Atterrissage d'un vaisseau spatial 101

7 Résolution des équations aux dérivées partielles 103
7.1 Introduction 103
7.2 Équation de diffusion thermique 104
7.2.1 La conduction thermique dans les solides 104
7.2.2 Notion de flux d'énergie 104
7.2.3 L'expression de la loi de Fourier 105
7.2.4 Comment obtenir cette équation ? 105
7.2.5 Résolution numérique de l'équation de diffusion thermique 107
7.3 L'équation de Laplace 110
7.4 Travaux dirigés 121
7.4.1 Exercice 1 : Équation de la chaleur 2D 121
7.4.2 Exercice 2 : Condensateur plan 122

Chapitre 1

Introduction à Python I : Présentation & installation

1.1 Objectifs généraux en premier

Une partie essentielle de ce cours est de vous permettre de faire de la science par des expériences numériques et de développer des projets qui vous permettent d'étudier des systèmes complexes. Le but est d'améliorer ce que nous appelons la pensée algorithmique.

Algorithme Un ensemble fini d'instructions non ambiguës qui, étant donné un ensemble de conditions initiales, peuvent être effectuées dans une séquence prescrite pour atteindre un certain but.

1.2 Situation standard que nous rencontrons quotidiennement

La situation standard que nous rencontrons presque tous les séances de cours :

- Théorie + expérience + simulation est presque la norme dans la recherche et l'industrie.
- Être capable de modéliser des systèmes complexes. Résoudre de vrais problèmes.
- Accent la compréhension des principes fondamentaux et des lois dans les sciences.
- Être capable de visualiser, présenter, discuter, interpréter et venir avec une analyse critique des résultats, et développer une attitude éthique saine pour son propre travail.
- Améliorer le raisonnement sur la méthode scientifique.

Une bonne présentation des résultats obtenus via de bons rapports scientifiques, aide à inclure tous les aspects ci-dessus.

1.3 Langage Python

Python est un langage de programmation moderne de haut niveau, orienté objet et d'usage général.

Caractéristiques générales de Python :

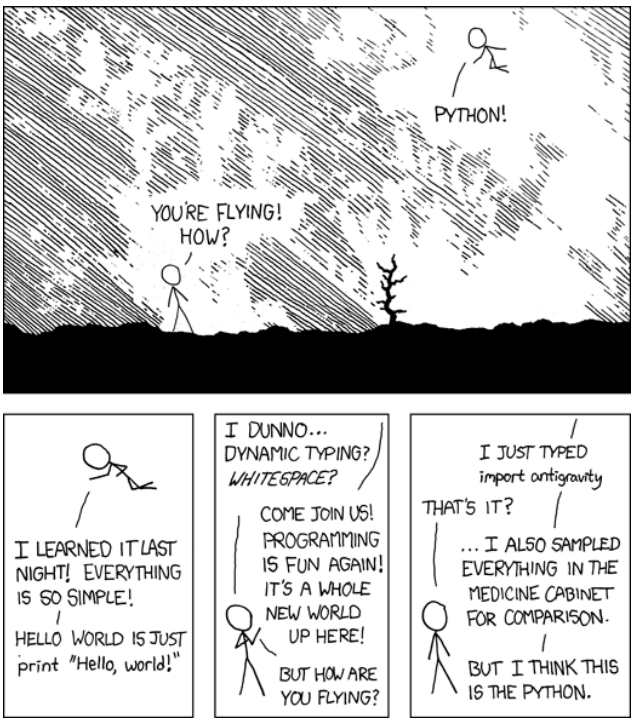
- Langage simple :
  - facile à lire et à apprendre avec une syntaxe minimaliste.
- Langage concis et expressif :
  - moins de lignes de code
  - moins de bugs
  - plus facile à maintenir.

Détails techniques :

- Typé dynamiquement :
  - Pas besoin de définir le type des variables, les arguments ou le type des fonctions.
- La gestion automatique de la mémoire :
  - Aucune nécessité d'allouer explicitement et désallouer la mémoire pour les variables et les tableaux de données. Aucun bug de fuite de mémoire.
- Interprété :
  - Pas besoin de compiler le code. L'interpréteur Python lit et exécute le code python directement.

Avantages :

- Le principal avantage est la facilité de programmation, qui minimise le temps nécessaire pour développer, déboguer et maintenir le code.
- Langage bien conçu qui encourage les bonnes pratiques de programmation :
  - Modulaire et orientée objet, permet l'encapsulation et la réutilisation de code. Il en résulte souvent un code plus transparent, plus facile à améliorer et sans bug.
  - Documentation intégré avec le code.
- De nombreuses bibliothèques standards, et de nombreux packages add-on.



1.4 Installation d'un environnement Python scientifique

1.4.1 Installation sur ordinateur

Qu'est ce que Anaconda ? L'installation d'un environnement Python complet peut-être une vraie galère. Déjà, il faut télécharger Python et l'installer. Par la suite, télécharger un à un les packages dont on a besoin. Parfois, le nombre de ces librairies peut-être grand.

Par ailleurs, il faut s'assurer de la compatibilité entre les versions des différentes packages qu'on a à télécharger. Bref, ce n'est pas amusant.

Anaconda est une distribution Python. A son installation, Anaconda installera Python ainsi qu'une multitude de packages (voir liste de packages anaconda). Cela nous évite de nous ruer dans les problèmes d'incompatibilités entre les différents packages.

Finalement, Anaconda propose un outil de gestion de packages appelé conda. Ce dernier permettra de mettre à jour et installer facilement les librairies dont on aura besoin pour nos développements.

Préparer la formation : téléchargement d'Anaconda. Nous demandons à tous les étudiants de télécharger Anaconda. Pour cela, il faut télécharger un installeur à partir de `https://www.anaconda.com/download/`, correspondant à votre système d'exploitation (Windows, Mac OS X, Linux). Il faut choisir entre 32 bits ou 64 bits (pour la version *Python 3*) selon que votre système d'exploitation est 32 bits ou 64 bits.

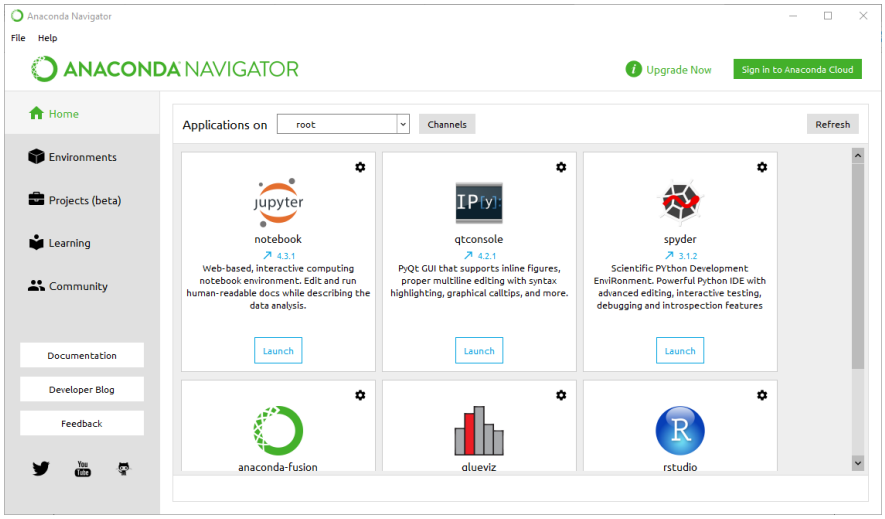


FIGURE 1.1 – Interface graphique du navigateur Anaconda sur Windows

> **Note:** Anaconda installe plusieurs exécutables pour développer en Python dans le répertoire *anaconda/bin*, sans toujours créer des raccourcis sur le bureau ou dans un menu. Nous nous occuperons au tout début de la formation de créer des raccourcis pour pouvoir lancer l'application web *Jupyter notebook*. Vous pouvez lancer le notebook depuis le navigateur Anaconda.

1.4.2 Spyder

Pour le développement de programmes en langage Python, des applications spéciales appelées IDE (Integrated Development Environment) peuvent être utilisées. Les IDE les plus avancés ont des éditeurs, des consoles, des outils pour organiser des suites de programmes et de bibliothèques, un correcteur orthographique (spell-checking) et une complétion automatiques (auto-completion) pour les scripts partiellement écrits (ces outils connaissent la syntaxe du langage de programmation) et des outils de débogage.

Utiliser un bon éditeur pour programmer en Python est bon. Utiliser un vrai IDE est encore plus confortable et puissant. **Spyder** (Scientific PYthon Development EnviRonment) semble actuellement très répandu pour l'utilisation scientifique de Python.

Spyder est un environnement de développement interactif gratuit inclus avec Anaconda. Il comprend des fonctionnalités d'édition, de test interactif, de débogage et d'introspection.

Après avoir installé Anaconda, vous pouvez démarrer Spyder sur macOS, Linux ou Windows en ouvrant une fenêtre de terminal (Ubuntu/macOS) ou d'invite de commande (Windows) et en exécutant la commande `spyder`.

1.4.3 Installation sur smartphone

Pydroid 3 - IDE éducatif pour Python 3. Pydroid 3 est l'IDE éducatif Python 3 le plus simple et le plus puissant à utiliser pour Android.

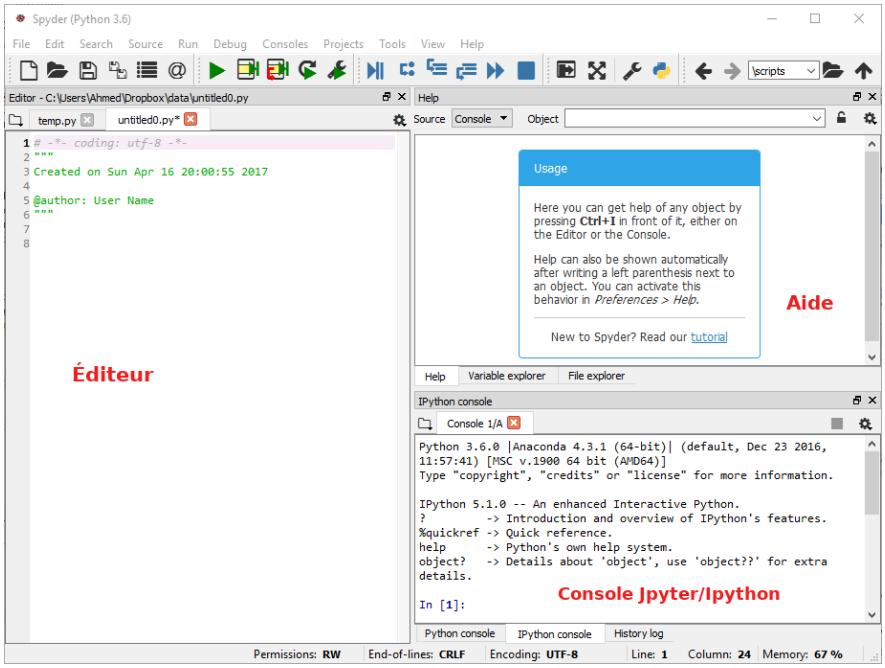


FIGURE 1.2 – Spyder sous Windows.

Pydroid 3 fournit :

- Interpréteur Python 3.6 hors connexion : Internet n'est pas nécessaire pour exécuter des programmes Python.
- Pip package manager et un référentiel personnalisé pour les packages de roues prédéfinis pour les bibliothèques scientifiques améliorées, tels que numpy, scipy, matplotlib, scikit-learn et jupyter.
- ...

Installer et utiliser Pydroid 3 sur son smartphone : Pydroid est une application Android que vous pouvez obtenir sur Google Play : `https://play.google.com/store/apps/details?id=ru.iiec.pydroid3`

Les étapes suivantes, dans les figures ci-dessous, vous permettent d'utiliser le cahier Jupyter sur votre téléphone portable n'importe où et à tout moment pour vous entraîner au maximum et vous familiariser avec tous les exemples de programmation de ce cours.

Phase installation :

1. Installer Pydroid 3 depuis Google Play : `https://play.google.com/store/apps/details?id=ru.iiec.pydroid3`

2. Ouvrez l'application, sur le menu cliquez sur pip et allez à l'onglet "QUICK INSTALL" pour obtenir les bibliothèques scientifiques nécessaires à ce cours.

3. Dans "QUICK INSTALL", installer les packages *Jupyter* , *numpy* et *matplotlib*.

Phase utilisation :

4. Retournez au menu et ouvrez le **terminal**.

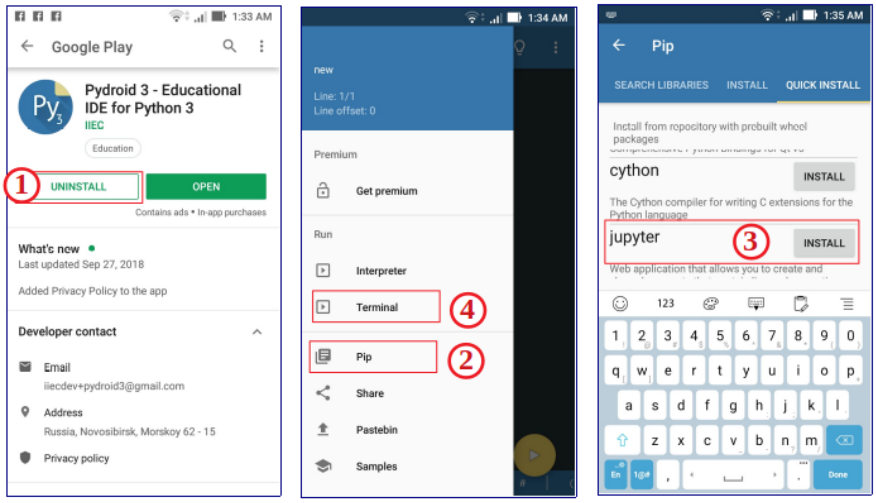


FIGURE 1.3 – Pydroid 3 : Phase installation

5. Sur le terminal, entrez la commande suivante :

```
jupyter notebook
```

6. Jupyter s'exécutera sur votre navigateur Web. Accédez au répertoire dans lequel vous avez des notebooks à ouvrir, à télécharger (bouton *upload*) ou à créer (bouton *New*).

7. Amusez-vous à travailler sur le notebook : créez du contenu, lancez et modifiez des exemples

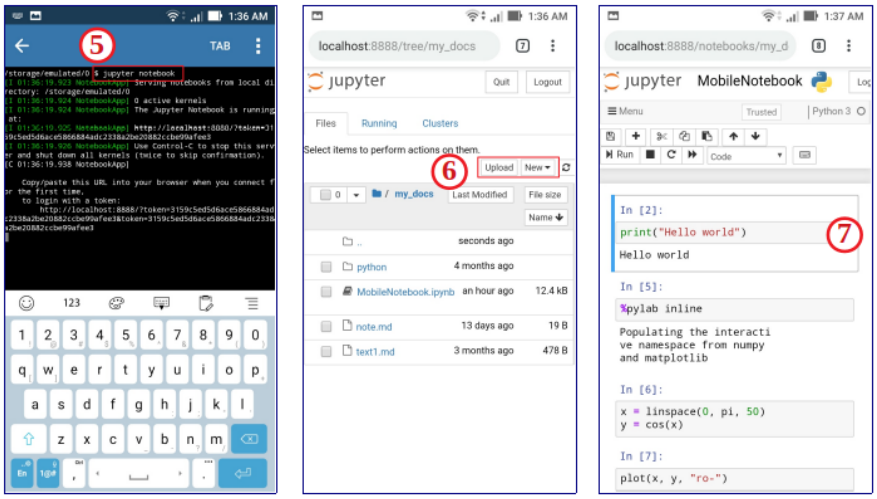


FIGURE 1.4 – Pydroid 3 : Phase utilisation

Chapitre 2

Introduction à Python II : syntaxe et variables

2.1 Introduction : "Hello World !"

C'est devenu une tradition que lorsque vous apprenez un nouveau langage de programmation, vous démarrez avec un programme permettant à l'ordinateur d'imprimer le message *"Hello World !"*.

```
In [1]: print("Hello World!")
Hello World!
```

Félicitation ! tout à l'heure vous avez fait votre ordinateur saluer le monde en anglais ! La fonction `print()` est utilisée pour imprimer l'instruction entre les parenthèses. De plus, l'utilisation de guillemets simples `print('Hello World!')` affichera le même résultat. Le délimiteur de début et de fin doit être le même.

```
In [2]: print('Hello World!')
Hello World!
```

2.2 Commentaires

Au fur et à mesure que vos programmes deviennent plus grands et plus compliqués, ils deviennent plus difficiles à lire et à regarder un morceau de code et à comprendre ce qu'il fait ou pourquoi. Pour cette raison, il est conseillé d'ajouter des notes à vos programmes pour expliquer en langage naturel ce qu'il fait. Ces notes s'appellent des commentaires et commencent par le symbole `#`.

Voyez ce qui se passe lorsque nous ajoutons un commentaire au code précédent :

```
In [3]: print('Hello World!') # Ceci est mon premier commentaire
Hello World!
```

Rien ne change dans la sortie ? Oui, et c'est très normal, l'interprète Python ignore cette ligne et ne renvoie rien. La raison en est que les commentaires sont écrits pour les humains, pour comprendre leurs codes, et non pour les machines.

2.3 Nombres

L'interpréteur Python agit comme une simple calculatrice : vous pouvez y taper une expression et l'interpréteur restituera la valeur. La syntaxe d'expression est simple : les opérateurs +, -, * et / fonctionnent comme dans la plupart des autres langages (par exemple, Pascal ou C) ; les parenthèses (`()`) peuvent être utilisées pour le regroupement. Par exemple :

```
In [4]: 5+3
Out[4]: 8
In [5]: 2 - 9      # les espaces sont optionnels
Out[5]: -7
In [6]: 7 + 3 * 4  #la hiérarchie des opérations mathématique
Out[6]: 19
In [7]: (7 + 3) * 4  # est-elle respectées?
Out[7]: 40
# en python3 la division retourne toujours un nombre en virgule flottante
In [8]: 20 / 3
Out[8]: 6.666666666666667
In [9]: 7 // 2      # une division entière
Out[9]: 3
```

On peut noter l'existence de l'opérateur `%` (appelé opérateur modulo). Cet opérateur fournit le reste de la division entière d'un nombre par un autre. Par exemple :

```
In [10]: 7 % 2       # donne le reste de la division
Out[10]: 1
In [11]: 6 % 2
Out[11]: 0
```

Les exposants peuvent être calculés à l'aide de doubles astérisques `**`.

```
In [12]: 3**2
Out[12]: 9
```

Les puissances de dix peuvent être calculées comme suit :

```
In [13]: 3 * 2e3   # vaut 3 * 2000
Out[13]: 6000.0
```

2.4 Affectations (ou assignation)

2.4.1 variables

Dans presque tous les programmes Python que vous allez écrire, vous aurez des variables. Les variables agissent comme des espaces réservés pour les données. Ils peuvent aider à court terme, ainsi qu'à la logique, les variables pouvant changer, d'où leur nom. C'est beaucoup plus facile en Python car aucune déclaration de variables n'est requise. Les noms de variable (ou tout autre objet Python tel que fonction, classe, module, etc.) commencent par une lettre majuscule ou minuscule (A-Z ou a-z). Ils sont sensibles à la casse (`VAR1` et `var1` sont deux variables distinctes). Depuis Python, vous pouvez utiliser n'importe quel caractère Unicode, il est préférable d'ignorer les caractères ASCII (donc pas de caractères accentués).

Si une variable est nécessaire, pensez à un nom et commencez à l'utiliser comme une variable, comme dans l'exemple ci-dessous :

Pour calculer l'aire d'un rectangle par exemple : `largeur` x `hauteur` :

```
In [15]: largeur = 25
    ...: hauteur = 40
    ...: largeur    # essayer d'accéder à la valeur de la variable largeur
Out[15]: 25
```

on peut également utiliser la fonction `print()` pour afficher la valeur de la variable `largeur`

```
In [16]: print(largeur)
25
```

Le produit de ces deux variables donne l'aire du rectangle :

```
In [17]: largeur * hauteur  # donne l'aire du rectangle
Out[17]: 1000
```

> **Note:** Notez ici que le signe égal (=) dans l'affectation ne doit pas être considéré comme **"est égal à"**. Il doit être **"lu"** ou interprété comme **"est définie par"**, ce qui signifie dans notre exemple :
>
> > La variable `largeur` est définie par la valeur 25 et la variable `hauteur` est définie par la valeur 40.

> **Note:** Si une variable n'est pas *définie* (assignée à une valeur), son utilisation vous donnera une erreur :

```
In [18]: aire     # essayer d'accéder à une variable non définie
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-18-1b03529c1ce5> in <module>()
----> 1 aire     # essayer d'accéder à une variable non définie

NameError: name 'aire' is not defined
```

Laissez-nous résoudre ce problème informatique (ou **bug** tout simplement) !. En d'autres termes, assignons la variable `aire` à sa valeur.

```
In [19]: aire = largeur * hauteur
    ...: aire  # et voila!
Out[19]: 1000
```

2.4.2 Noms de variables réservés (keywords)

Certains noms de variables ne sont pas disponibles, ils sont réservés à python lui-même. Les mots-clés suivants (que vous pouvez afficher dans l'interpréteur avec la commande `help("keywords")`) sont réservés et ne peuvent pas être utilisés pour définir vos propres identifiants (variables, noms de fonctions, classes, etc.).

```
In [20]: help("keywords")

Here is a list of the Python keywords.  Enter any keyword to get more help.

False               def                 if                  raise
None                del                 import              return
True                elif                in                  try
and                 else                is                  while
as                  except              lambda              with
assert              finally             nonlocal            yield
break               for                 not
class               from                or
continue            global              pass

# par exemple pour éviter d'écraser le nom réservé lambda
In [22]: lambda_ = 630e-9
    ...: lambda_
Out[22]: 6.3e-07
```

2.4.3 Les types

Les types utilisés dans Python sont : integers, long integers, floats (double prec.), complexes, strings, booleans. La fonction `type()` donne le type de son argument

Le type int (integer : nombres entiers). Pour affecter (on peut dire aussi assigner) la valeur 20 à la variable nommée `age` :

```
age = 20
```

La fonction `print()` affiche la valeur de la variable :

```
In [24]: print(age)
20
```

La fonction `type()` retourne le type de la variable :

```
type(age)
Out[25]: int
```

Le type float (nombres en virgule flottante).

```
b = 17.0  # le séparateur décimal est un point (et non une virgule)
b
Out[26]: 17.0
In [27]: type(b)
Out[27]: float
In [28]: c = 14.0/3.0
    ...: c
Out[28]: 4.666666666666667
```

Notation scientifique :

```
In [29]: a = -1.784892e4
    ...: a
Out[29]: -17848.92
```

Les fonctions mathématiques. Pour utiliser les fonctions mathématiques, il faut commencer par importer le module `math` :

```
import math
```

La fonction `help()` retourne la liste des fonctions et données d'un module.

Soit par exemple : `help('math')`

Pour appeler une fonction d'un module, la syntaxe est la suivante : `module.fonction(arguments)`

Pour accéder à une donnée d'un module : `module.data`

```
 # donnée pi du module math (nombre pi)
In [32]: math.pi
Out[32]: 3.141592653589793
# fonction sin() du module math (sinus)
In [33]: math.sin(math.pi/4.0)
Out[33]: 0.7071067811865475
# fonction sqrt() du module math (racine carrée)
In [34]: math.sqrt(2.0)
Out[34]: 1.4142135623730951
# fonction exp() du module math (exponentielle)
In [35]: math.exp(-3.0)
Out[35]: 0.049787068367863944
# fonction log() du module math (logarithme népérien)
In [36]: math.log(math.e)
Out[36]: 1.0
```

Le type complexe. Python possède par défaut un type pour manipuler les nombres complexes. La partie imaginaire est indiquée grâce à la lettre « `j` » ou « `J` ». La lettre mathématique utilisée habituellement, le « `i` », n'est pas utilisée en Python car la variable i est souvent utilisée dans les boucles.

```
In [37]: a = 2 + 3j
    ...: type(a)
Out[37]: complex
In [38]: a
Out[38]: (2+3j)
```

Note:

```
In [39]: b = 1 + j
---------------------------------------------------------------
NameError                         Traceback (most recent call last)
<ipython-input-39-0f22d953f29e> in <module>()
----> 1 b = 1 + j

NameError: name 'j' is not defined
```

Dans ce cas, on doit écrire la variable `b` comme suit :

```
In [41]: b = 1 + 1j
    ...: b
Out[41]: (1+1j)
```

sinon Python va considérer `j` comme variable non définie.

On peut faire l'addition des variables complexes :

```
In [42]: a + b
Out[42]: (3+4j)
```

Le type str (string : chaîne de caractères).

```
In [43]: nom = 'Tounsi' # entre apostrophes
    ...: nom
Out[43]: 'Tounsi'
In [44]: type(nom)
Out[44]: str
In [45]: prenom = "Ali"  # on peut aussi utiliser les guillemets
    ...: prenom
Out[45]: 'Ali'
In [46]: print(nom, prenom)  # ne pas oublier la virgule
Tounsi Ali
```

La concaténation désigne la mise bout à bout de plusieurs chaînes de caractères. La concaténation utilise l'opérateur `+` :

```
In [47]: chaine = nom + prenom  # concaténation de deux chaînes de caractères
    ...: chaine
Out[47]: 'TounsiAli'
```

Vous voyez dans cet exemple que le nom et le prénom sont collé. Pour ajouter une espace entre ces deux chaînes de caractères :

```
In [48]: chaine = prenom + ' ' + nom
    ...: chaine # et voila
Out[48]: 'Ali Tounsi'
```

On peut modifier/ajouter une nouvelle chaîne à notre variable `chaine` par :

```
In [49]: chaine = chaine + ' 22 ans'  # en plus court : chaine += ' 22 ans'
    ...: chaine
Out[49]: 'Ali Tounsi 22 ans'
```

La fonction `len()` renvoie la longueur (*length*) de la chaîne de caractères :

```
In [53]: print(nom)
    ...: len(nom)
Tounsi
Out[53]: 6
```

Indexage et slicing :

```
 +---+---+---+---+---+---+
|-----------------------|
 | T | o | u | n | s | i |
 +---+---+---+---+---+---+
 |-----------------------|
 0   1   2   3   4   5   6
 --->
-6  -5  -4  -3  -2  -1
                 <----
```

```
In [55]: nom[0]  # premier caractère (indice 0)
Out[55]: 'T'

In [56]: nom[:] # toute la chaine
Out[56]: 'Tounsi'

In [57]: nom[1] # deuxième caractère (indice 1)
Out[57]: 'o'

In [58]: nom[1:4]   # slicing
Out[58]: 'oun'

In [59]: nom[2:]  # slicing
Out[59]: 'unsi'

In [60]: nom[-1]   # dernier caractère (indice -1)
Out[60]: 'i'

In [61]: nom[-3:]    # slicing
Out[61]: 'nsi'
```

Note:

On ne peut pas mélanger le type `str` et type `int`.

Soit par exemple :

```
In [63]: chaine = '22'
    ...: annee_naissance = 2018 - chaine
---------------------------------------------------------------
TypeError                 Traceback (most recent call last)
<ipython-input-63-8607078f78d2> in <module>()
      1 chaine = '22'
----> 2 annee_naissance = 2018 - chaine

TypeError: unsupported operand type(s) for -: 'int' and 'str'
```

Pour corriger cette erreur, la fonction `int()` permet de convertir un type `str` en type `int` :

```
In [64]: nombre = int(chaine)
    ...: type(nombre) # et voila!
Out[64]: int
```

Maintenant on peut trouver `annee_naissance` sans aucun problème :

```
In [65]: annee_naissance = 2018 - nombre
    ...: annee_naissance
Out[65]: 1996
```

Interaction avec l'utilisateur (la fonction `input()`) La fonction `input()` lance une case pour saisir une chaîne de caractères.

```
In [66]: prenom = input('Entrez votre prénom : ')
    ...: age = input('Entrez votre age : ')

Entrez votre prénom : Foulen

Entrez votre age : 25
```

Formatage des chaînes Un problème qui se retrouve souvent, c'est le besoin d'afficher un message qui contient des valeurs de variables.

Soit le message : Bonjour Mr/Mme `prenom`, votre age est `age`.

La solution est d'utiliser la méthode `format()` de l'objet chaîne `str()` et le `{}` pour définir la valeur à afficher.

```
print(" Bonjour Mr/Mme {}, votre age est {}.".format(prenom, age))
```

Le type list (liste) Une liste est une structure de données.

Le premier élément d'une liste possède l'indice (l'index) 0.

Dans une liste, on peut avoir des éléments de plusieurs types.

```
In [1]: info = ['Tunisie', 'Afrique', 3000, 36.8, 10.08]

In [2]: type(info)
Out[2]: list
```

La liste info contient 5 éléments de types str, str, int, float et float

```
In [3]: info
Out[3]: ['Tunisie', 'Afrique', 3000, 36.8, 10.08]

In [4]: print('Pays : ', info[0])    # premier élément (indice 0)
Pays :  Tunisie

In [5]: print('Age : ', info[2])     # le troisième élément a l'indice 2
Age :  3000

In [6]: print('Latitude : ', info[3]) # le quatrième élément a l'indice 3
Latitude :  36.8
```

La fonction `range()` crée une liste d'entiers régulièrement espacés :

```
In [7]: maliste = range(10) # équivalent à range(0,10,1)
   ...: type(maliste)
Out[7]: range
```

Pour convertir une range en une liste, on applique la fonction `list()` à notre variable :

```
In [8]: list(maliste)   # pour convertir range en une liste
Out[8]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

On peut spécifier le début, la fin et l'intervalle d'une range :

```
In [9]: maliste = range(1,10,2)   # range(début,fin non comprise,intervalle)
   ...: list(maliste)
Out[9]: [1, 3, 5, 7, 9]

In [10]: maliste[2] # le troisième élément a l'indice 2
Out[10]: 5
```

On peut créer une liste de listes, qui s'apparente à un tableau à 2 dimensions (ligne, colonne) :

```
0   1   2
10  11  12
20  21  22



In [11]: maliste = [[0, 1, 2], [10, 11, 12], [20, 21, 22]]
    ...: maliste[0]
Out[11]: [0, 1, 2]

In [12]: maliste[0][0]
Out[12]: 0

In [13]: maliste[2][1] # élément à la troisième ligne et deuxième colonne
Out[13]: 21

In [14]: maliste[2][1] = 78   # nouvelle affectation

In [15]: maliste
Out[15]: [[0, 1, 2], [10, 11, 12], [20, 78, 22]]
```

Le type bool (booléen). Deux valeurs sont possibles : `True` et `False`

```
In [16]: choix = True # NOTE: "True" différent de "true"
    ...: type(choix)
Out[16]: bool
```

Les opérateurs de comparaison :

Opérateur	Signification	Remarques
`<`	strictement inférieur	
`<=`	inférieur ou égal	
`>`	strictement supérieur	
`>=`	supérieur ou égal	
`==`	égal	Attention : deux signes `==`
`!=`	différent	

```
In [17]: b = 10
    ...: b > 8
Out[17]: True

In [18]: b == 5
Out[18]: False

In [19]: b != 5
Out[19]: True

In [20]: 0 <= b <= 20
Out[20]: True
```

Les opérateurs logiques : `and`, `or`, `not`

```
In [21]: note = 13.0

In [22]: mention_ab = note >= 12.0 and note < 14.0

In [23]: # ou bien : mention_ab = 12.0 <= note < 14.0

In [24]: mention_ab
Out[24]: True


In [25]: not mention_ab
Out[25]: False

In [26]: note == 20.0 or note == 0.0
Out[26]: False
```

L'opérateur `in` s'utilise avec des chaînes (type `str`) ou des listes (type `list`).

Pour une chaînes :

```
In [30]: chaine = 'Bonsoir'
    ...: #la sous-chaîne 'soir' fait-elle partie de la chaîne 'Bonsoir' ?

In [31]: resultat = 'soir' in chaine
    ...: resultat
Out[31]: True
```

Pour une liste :

```
In [32]: maliste = [4, 8, 15]
    ...: #le nombre entier 9 est-il dans la liste ?

In [33]: 9 in maliste
Out[33]: False

In [34]: 8 in maliste
Out[34]: True

In [35]: 14 not in maliste
Out[35]: True
```

2.5 Lectures complémentaires

— Documentation Python 3.6 : `https://docs.python.org/fr/3.6/tutorial/index.html`
— Apprendre à programmer avec Python, par Gérard Swinnen : `http://inforef.be/swi/python.htm`
— Think Python, par Allen B. Downey : `https://greenteapress.com/wp/think-python/`

2.6 Travaux dirigés

2.6.1 Exercice 1 : Les variables

a) Assignez les valeurs respectives 3, 5, 7 à trois variables a, b, c. Effectuez l'opération `a-b//c`. Interprétez le résultat obtenu.

b) Testez les lignes d'instructions suivantes. Décrivez ce qui se passe :

```
r = 12
pi = 3.14159
s = pi * r**2
print(s)
print(type(r), type(pi), type(s))
```

Quelle est, à votre avis, l'utilité de la fonction `type()` ?

c) Écrivez un programme qui convertisse en degrés Celsius une température exprimée au départ en degrés Fahrenheit, ou l'inverse. La formule de conversion est :

$$T_F = T_C \times 1,8 + 32$$

2.6.2 Exercice 2 : Fonction `input()`

Dans tous ces exercices, utilisez la fonction `input()` pour l'entrée des données.

a) Écrivez un programme qui convertisse en mètres par seconde et en km/h une vitesse fournie par l'utilisateur en miles/heure. (Rappel : 1 mile = 1609 mètres)

b) Écrivez un programme qui calcule le périmètre et l'aire d'un triangle quelconque dont l'utilisateur fournit les 3 côtés.(Rappel : l'aire d'un triangle quelconque se calcule à l'aide de la formule :

$$S = \sqrt{d \cdot (d-a) \cdot (d-b) \cdot (d-c)}$$

dans laquelle d désigne la longueur du demi-périmètre, et a, b, c celles des trois côtés.)

2.6.3 Exercice 3 : Corriger l'erreur dans le code

a) Le code suivant renvoie une erreur. Trouver et corriger l'erreur :

```
prenom = input('Entrez votre prénom : ')
age = input('Entrez votre age : ')
annee_naissance = 2018 - age
print("Bonjour Mr/Mme", prenom)
print("vous êtes né en", annee_naissance)
```

Indication.

— utiliser la fonction `type()`

— utliser la fonction `int()`

b) Afficher le messge suivant : Bonjour Mr/Mme `prenom`, votre age est `age` et vous êtes né en `annee_naissance`.

Indication. Remplacer les points par ce qui convient dans le code :

```
print(" Bonjour Mr/Mme {}, votre age est {}....".format(...))
```

Chapitre 3

Introduction à Python III : Contrôle du flux d'instructions

3.1 Les conditions

3.1.1 L'instruction `if`

En programmation, nous avons toujours besoin de la notion de condition pour permettre à un programme de s'adapter à différents cas de figure.

Syntaxe

```
if expression: # ne pas oublier le signe de ponctuation ':'
    "bloc d'instructions" # attention à l'indentation (1 Tab ou 4 * Espaces)
# suite du programme
```

- Si l'expression est vraie (`True`) alors le bloc d'instructions est exécuté.
- Si l'expression est fausse (`False`) on passe directement à la suite du programme.

Exemple 1 : Note sur 20. Dans cet exemple nous allons tester si la note entrée par l'utilisateur. Si la note est > 10 on doit recevoir le message : "J'ai la moyenne" sinon il va rien faire.

```
chaine = input("Note sur 20 : ")
note = float(chaine)
if note >= 10.0:
    # ce bloc est exécuté si l'expression (note >= 10.0) est vraie
    print("J'ai la moyenne")

# suite du programme
print("Fin du programme")
```

Note:

- Les blocs de code sont délimités par l'indentation.

— L'indentation est obligatoire dans les scripts.

3.1.2 L'instruction `else`

Une instruction `else` est toujours associée à une instruction `if`.

Syntaxe

```
if expression:
    "bloc d'instructions 1"    # attention à l'indentation (1 Tab ou 4 * Espaces)
else:                          # else est au même niveau que if
    "bloc d'instructions 2"    # attention à l'indentation
# suite du programme
```

— Si l'expression est vraie (`True`) alors le bloc d'instructions 1 est exécuté.
— Si l'expression est fausse (`False`) alors c'est le bloc d'instructions 2 qui est exécuté.

Exemple 2 : moyenne. Dans cet exemple nous allons tester si la note entrée par l'utilisateur. Si la note est > 10 on doit recevoir le message : "J'ai la moyenne" sinon il va afficher "C'est en dessous de la moyenne".

```
chaine = input("Note sur 20 : ")
note = float(chaine)
if note >= 10.0:
    # ce bloc est exécuté si l'expression (note >= 10.0) est vraie
    print("J'ai la moyenne")
else:
    # ce bloc est exécuté si l'expression (note >= 10.0) est fausse
    print("C'est en dessous de la moyenne")
print("Fin du programme")
```

Pour traiter le cas des notes invalides (< 0 ou > 20), on peut imbriquer des instructions conditionnelles :

```
chaine = input("Note sur 20 : ")
note = float(chaine)
if note > 20.0 or note < 0.0:
    # ce bloc est exécuté si l'expression (note > 20.0 or note < 0.0) est vraie
    print("Note invalide !")
else:
    # ce bloc est exécuté si l'expression (note > 20.0 or note < 0.0) est fausse
    if note >= 10.0:
        # ce bloc est exécuté si l'expression (note >= 10.0) est vraie
        print("J'ai la moyenne")
    else:
        # ce bloc est exécuté si l'expression (note >= 10.0) est fausse
        print("C'est en dessous de la moyenne")
print("Fin du programme")
```

Ou bien encore :

```
chaine = input("Note sur 20 : ")
note = float(chaine)
if note > 20.0 or note < 0.0:
    print("Note invalide !")
else:
    if note >= 10.0:
```

```
        print("J'ai la moyenne")
        if note == 20.0:
            # ce bloc est exécuté si l'expression (note == 20.0) est vraie
            print("C'est même excellent !")
    else:
        print("C'est en dessous de la moyenne")
        if note == 0.0:
            # ce bloc est exécuté si l'expression (note == 0.0) est vraie
            print("... lamentable !")
print("Fin du programme")
```

3.1.3 L'instruction `elif`

Une instruction `elif` (contraction de **else if**) est toujours associée à une instruction `if`.

Syntaxe

```
if expression 1:
    "bloc d'instructions 1"
elif expression 2:
    "bloc d'instructions 2"
elif expression 3:
    "bloc d'instructions 3"    # ici deux instructions elif, mais il n'y a pas de limitation
else:
    "bloc d'instructions 4"
# suite du programme
```

- Si l'expression 1 est vraie alors le bloc d'instructions 1 est exécuté, et on passe à la suite du programme.
- Si l'expression 1 est fausse alors on teste l'expression 2 :
- si l'expression 2 est vraie on exécute le bloc d'instructions 2, et on passe à la suite du programme.
- si l'expression 2 est fausse alors on teste l'expression 3, etc.

Le bloc d'instructions 4 est donc exécuté si toutes les expressions sont fausses (c'est le bloc "par défaut").

Parfois il n'y a rien à faire. Dans ce cas, on peut omettre l'instruction `else` :

```
if expression 1:
    "bloc d'instructions 1"
elif expression 2:
    "bloc d'instructions 2"
elif expression 3:
    "bloc d'instructions 3"
# suite du programme
```

L'instruction `elif` évite souvent l'utilisation de conditions imbriquées (et souvent compliquées).

Exemple 3 : moyenne-bis. On peut tester plusieurs possibilités avec une syntaxe beaucoup plus propre avec les instructions `if-elif-else` :

```
note = float(input("Note sur 20 : "))
if note == 0.0:
    print("C'est en dessous de la moyenne")
```

```
    print("... lamentable!")
elif note == 20.0:
    print("J'ai la moyenne")
    print("C'est même excellent !")
elif 0 < note < 10:    # ou bien : elif 0.0 < note < 10.0:
    print("C'est en dessous de la moyenne")
elif note >= 10.0 and note < 20.0:   # ou bien : elif 10.0 <= note < 20.0:
    print("J'ai la moyenne")
else:
    print("Note invalide !")
print("Fin du programme")
```

3.1.4 Exercice : Condition sur le jour de travail

Si aujourd'hui est lundi alors je dois aller travailler, mais si c'est dimanche alors je peux rester faire la grasse matinée. Pour pouvoir accomplir ce genre de choses en Python, on fait appel à des expressions booléennes qui ne peuvent revêtir que deux possibilités - ou bien l'expression est vraie ou bien elle est fausse - et à la syntaxe if condition : qui permet de contrôler le flux du programme grâce à ces valeurs booléennes.

```
day_week = ["Lundi", "Mardi", "Mercredi", "Jeudi", "Vendredi",
            "Samedi", "Dimanche"]
today = input("Aujourd'hui est: ")

if *condition vraie*:  # Quelle est la condition vraie dans ce cas?
    print("Je dors le matin!")
else:
    print("Je travail le matin!")

print("Fin du programme")
```

Indication. Dans la **condition vraie**, utilisez l'opérateur logique `in` pour tester les éléments de la liste `day_week`.

Solution.

```
day_week = ["Lundi", "Mardi", "Mercredi", "Jeudi", "Vendredi", "Samedi", "Dimanche"]
today = input("Aujourd'hui est: ")
if today in day_week:
    if today == day_week[-1]:  # la condition vraie
        print("Je dors le matin!")
    else:
        print("Je travail le matin!")

print("Fin du programme")
```

3.2 Les boucles

3.2.1 L'instruction `while`

Syntaxe

```
while expression:           # ne pas oublier le signe de ponctuation ':'
    "bloc d'instructions"   # attention à l'indentation (1 Tab ou 4 * Espaces)
# suite du programme
```

— Si l'expression est vraie (`True`) le bloc d'instructions est exécuté, puis l'expression est à nouveau évaluée.
— Le cycle continue jusqu'à ce que l'expression soit fausse (`False`) : on passe alors à la suite du programme.

Exemple 1 : un script qui compte de 1 à 4.

```
# initialisation de la variable de comptage
compteur = 0
while compteur < 5:
    # ce bloc est exécuté tant que la condition (compteur < 5) est vraie
    print(compteur)
    compteur +=  1    # incrémentation du compteur,  compteur = compteur + 1
print(compteur)
print("Fin de la boucle")
```

Exemple 2 : Table de multiplication par 8.

```
compteur = 1          # initialisation de la variable de comptage
while compteur <= 10:
    # ce bloc est exécuté tant que la condition (compteur <= 10) est vraie
    print(compteur, '* 8 =', compteur*8)
    compteur += 1     # incrémentation du compteur, compteur = compteur + 1
print("Et voilà !")
```

Exemple 3 : Affichage de l'heure courante.

```
import time      # importation du module time
quitter = 'n'    # initialisation
while quitter != 'o':
    # ce bloc est exécuté tant que la condition est vraie
    # strftime() est une fonction du module time
    print('Heure courante ', time.strftime('%H:%M:%S'))
    quitter = input("Voulez-vous quitter le programme (o/n) ? ")
print("A bientôt")
```

3.2.2 L'instruction `for`

Syntaxe

```
for élément in séquence :     # ne pas oublier le signe de ponctuation ':'
    "bloc d'instructions"     # attention à l'indentation (1 Tab ou 4 * Espaces)
# suite du programme
```

Les éléments de la séquence sont issus d'une chaîne de caractères ou bien d'une liste.

Exemple 1 : séquence de caractères.

```
chaine = 'Bonsoir'
for lettre in chaine:  # lettre est la variable d'itération
    print(lettre)
print("Fin de la boucle")
```

La variable lettre est initialisée avec le premier élément de la séquence ('B'). Le bloc d'instructions est alors exécuté.

Puis la variable lettre est mise à jour avec le second élément de la séquence ('o') et le bloc d'instructions à nouveau exécuté...

Le bloc d'instructions est exécuté une dernière fois lorsqu'on arrive au dernier élément de la séquence ('r').

Fonction `range()`. L'association avec la fonction `range()` est très utile pour créer des séquences automatiques de nombres entiers :

```
for i in range(1, 5):
    print(i)
print("Fin de la boucle")
```

Exemple 2 : Table de multiplication. La création d'une table de multiplication paraît plus simple avec une boucle `for` qu'avec une boucle `while` :

```
for compteur in range(1,11):
    print(compteur, '* 8 =', compteur*8)
print("Et voilà !")
```

Exemple 3 : calcul d'une somme. Soit, par exemple, l'expression de la somme suivante :

$$s = \sum_{i=0}^{100} \sqrt{\frac{i\pi}{100}} sin(\frac{i\pi}{100})$$

```
from math import sqrt, sin, pi
s = 0.0 # # intialisation de s
for i in range(101):
    s+= sqrt(i * pi/100) * sin(i * pi/100)   # équivalent à s = s + sqrt(x) * sin(x)
# Affichage de la somme
print(s)
```

3.2.3 Exercice : produit de Wallis

Calculer π avec le produit de Wallis

$$\frac{\pi}{2} = \prod_{i=1}^{p} \frac{4i^2}{4i^2 - 1}$$

Solution.

```
# %load wallis.py
from math import pi

my_pi = 1.  # intialisation
p = 100000
for i in range(1, p):
    my_pi *= 4 * i ** 2 / (4 * i ** 2 - 1.)  # implémentation de la formule de Wallis

my_pi *= 2       # multiplication par 2 de la valeur trouvée

print("La valeur de pi de la bibliothèque 'math': ", pi)
print("La valeur de pi calculer par la formule de Wallis: ", my_pi)

print("La différence entre les deux valeurs:", abs(pi - my_pi))
# la fonction abs() donne la valeur absolue
```

3.2.4 Compréhensions de listes

Les compréhensions de listes fournissent un moyen de construire des listes de manière très concise. Une application classique est la construction de nouvelles listes où chaque élément est le résultat d'une opération appliquée à chaque élément d'une autre séquence ; ou de créer une sous-séquence des éléments satisfaisant une condition spécifique.

Par exemple, supposons que l'on veuille créer une liste de carrés, comme :

```
squares = []
for x in range(10):
    squares.append(x**2)


squares
```

Notez que cela crée (ou remplace) une variable nommée `x` qui existe toujours après l'exécution de la boucle. On peut calculer une liste de carrés sans effet de bord avec :

```
squares = [x**2 for x in range(10)]
squares
```

qui est plus court et lisible.

Une compréhension de liste consiste à placer entre crochets une expression suivie par une clause `for` puis par zéro ou plus clauses `for` ou `if`. Le résultat est une nouvelle liste résultat de l'évaluation de l'expression dans le contexte des clauses `for` et `if` qui la suivent. Par exemple, cette compréhension de liste combine les éléments de deux listes s'ils ne sont pas égaux :

```
combs = [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]


combs
```

et c'est équivalent à :

```
combs = []
for x in [1,2,3]:
    for y in [3,1,4]:
        if x != y:
            combs.append((x, y))


combs
```

> **Note:** Notez que l'ordre des instructions `for` et `if` est le même dans ces différents extraits de code.

3.2.5 L'instruction break

L'instruction break provoque une sortie immédiate d'une boucle `while` ou d'une boucle `for`.

Dans l'exemple suivant, l'expression `True` est toujours ... vraie : on a une boucle sans fin.

L'instruction `break` est donc le seul moyen de sortir de la boucle.

Exemple : Affichage de l'heure courante.

```
import time     # importation du module time
while True:
    # strftime() est une fonction du module time
    print('Heure courante ', time.strftime('%H:%M:%S'))
    quitter = input('Voulez-vous quitter le programme (o/n) ? ')
    if quitter == 'o':
        break
print("A bientôt")
```

> **Note:** Si vous connaissez le nombre de boucles à effectuer, utiliser une boucle `for`. Autrement, utiliser une boucle `while` (notamment pour faire des boucles sans fin).

3.3 Les fonctions

Nous avons déjà vu beaucoup de fonctions : `print()`, `type()`, `len()`, `input()`, `range()`...

Ce sont des fonctions pré-définies (Fonctions natives).

Nous avons aussi la possibilité de créer nos propres fonctions !

3.3.1 Intérêt des fonctions

Une fonction est une portion de code que l'on peut appeler au besoin (c'est une sorte de sous-programme).

L'utilisation des fonctions évite des redondances dans le code : on obtient ainsi des programmes plus courts et plus lisibles.

Par exemple, nous avons besoin de convertir à plusieurs reprises des degrés Celsius en degrés Fahrenheit :

$$T_F = T_C \times 1,8 + 32$$

```
print(100 * 1.8 + 32.0)


print(37.0 * 1.8 + 32.0)


print(233.0 * 1.8 + 32.0)
```

La même chose en utilisant une fonction :

```
def fahrenheit(degre_celsius):
        """
        Conversion degré Celsius en degré Fahrenheit
        """
        print(degre_celsius * 1.8 + 32.0)


fahrenheit(100)


fahrenheit(37)


temperature = 220
fahrenheit(temperature)
```

3.3.2 L'instruction def

Syntaxe

```
def nom_de_la_fonction(parametre1, parametre2, parametre3, ...):
    """
    Documentation
    qu'on peut écrire
    sur plusieurs lignes
    """     # docstring entouré de 3 guillemets (ou apostrophes)

    "bloc d'instructions"     # attention à l'indentation

    return resultat            # la fonction retourne le contenu de la variable resultat
```

Exemple : ma première fonction.

```
def mapremierefonction():         # cette fonction n'a pas de paramètre
    """
    Cette fonction affiche 'Bonjour'
    """
    print("Bonjour")
    return                        # cette fonction ne retourne rien ('None')
    # l'instruction return est ici facultative


mapremierefonction()


help(mapremierefonction)
```

3.4 Les Scripts

Commençons par écrire un script, c'est-à-dire un fichier avec une séquence d'instructions à exécuter chaque fois que le script est appelé. Les instructions peuvent être, par exemple, copié-collé depuis une **cellule code** dans votre notebook (mais veillez à respecter les règles d'indentation !).

L'extension pour les fichiers Python est `.py`. Écrivez ou copiez et collez les lignes suivantes dans un fichier appelé `test.py`

```
chaine = 'Bonsoir'
for lettre in chaine:  # lettre est la variable d'itération
    print(lettre)
```

Exécutons maintenant le script de manière interactive, à l'intérieur de l'interpréteur Ipython (cellule code du notebook). C'est peut-être l'utilisation la plus courante des scripts en calcul et simulation scientifique.

> **Note:** Dans la cellule *code* (Ipython), la syntaxe permettant d'exécuter un script est `%run script.py`. Par exemple :

```
%run test.py


chaine
```

La syntaxe permettant de charger le contenu d'un script dans dans une cellule code est `%load script.py`. Par exemple :

```
# %load test.py
chaine = 'Bonsoir'
for lettre in chaine:  # lettre est la variable d'itération
    print(lettre)
```

3.5 Lectures complémentaires

— Documentation Python 3.6
— Apprendre à programmer avec Python, par Gérard Swinnen
— Think Python, par Allen B. Downey

3.6 Travaux dirigés

3.6.1 Exercice 1 : Comparer deux entiers

Écrivez un programme qui vous demande de saisir 2 nombres entiers et affiche la plus petite de ces valeurs.

3.6.2 Exercice 2 : Comparer deux chaînes

Écrivez un programme qui demande d'entrer 2 chaînes et qui affiche la plus grande des 2 chaînes (celle qui contient le plus de caractères).

3.6.3 Exercice 3 : Convertir Euro contre Dinar Tunisien | EUR TND

Écrivez un programme qui convertit l'euro (EUR) en dinar tunisien (TND) :

— Le programme commence par demander à l'utilisateur d'indiquer par une chaîne de caractères 'EUR' ou 'TND' la devise du montant qu'il entrera.
— Ensuite, le programme exécute une action conditionnelle de la forme :

```
if devise == 'EUR' :
    # Expression 1
elif devise == 'TND' :
    # Expression 2
else :
   # affichage d'un message d'erreur
```

3.6.4 Exercice 4 : Résolution d'une équation du second degré

Soit l'équation du second degré $ax^2 + bx + c = 0$ où a, b et c sont des coefficients réels.

a) Écrivez un programme qui demande d'entrer les coefficients et affiche les solutions de l'équation.

Indication. Solutions analytiques

Des solutions sont recherchées dans le cas général, compte tenu du discriminant $\Delta = b^2 - 4ac$, l'équation admet comme solutions analytiques :

$$\begin{cases} \Delta > 0 & \textit{deux solutions réelles} : \ x_1 = \frac{-b-\sqrt{\Delta}}{2a}; \quad x_2 = \frac{-b+\sqrt{\Delta}}{2a} \\ \Delta = 0 & \textit{une solution double} : \ x_0 = \frac{-b}{2a} \\ \Delta < 0 & \textit{deux solutions complexes} : \ z_1 = \frac{-b-i\sqrt{-\Delta}}{2a}; \quad z_2 = \frac{-b+i\sqrt{-\Delta}}{2a} \end{cases}$$

Algorithme

Définition

Ensemble de règles opératoires dont l'application permet de résoudre un problème énoncé au moyen d'un nombre fini d'opérations. Un algorithme peut être traduit, grâce à un langage de programmation, en un programme exécutable par un ordinateur. Source: LAROUSSE

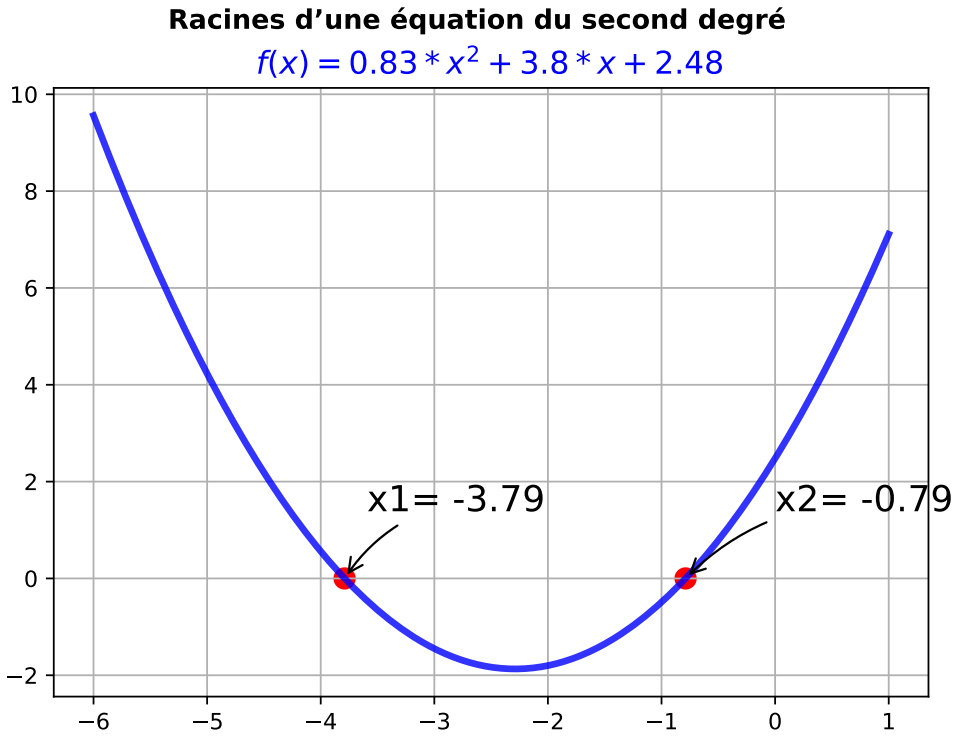
Pseudo-code de l'algorithme

Présentons tout d'abord un pseudo-code de l'algorithme, c'est-à-dire le détail des opérations à effectuer sans syntaxe propre du langage.

 # Calcul des racines de l'équation du second degré
a, b et c ← ... *# Assignation des variables a, b et c (variables de type réel) en utilisant la fonction* ***input()***
$\Delta \leftarrow b^2 - 4ac$
si Δ est positive:
␣␣␣␣ $x_1 \leftarrow \frac{-b-\sqrt{\Delta}}{2a}$
␣␣␣␣ $x_2 \leftarrow \frac{-b+\sqrt{\Delta}}{2a}$
␣␣␣␣*# Affichez les solutions trouvées*
sinon si Δ est nul:
␣␣␣␣ $x_0 \leftarrow \frac{-b}{2a}$
␣␣␣␣*# Affichez la solution trouvée*
sinon si Δ est négative:
␣␣␣␣ $z_1 \leftarrow \frac{-b-i\sqrt{-\Delta}}{2a}$
␣␣␣␣ $z_2 \leftarrow \frac{-b+i\sqrt{-\Delta}}{2a}$
␣␣␣␣*# Affichez les solutions trouvées*

b) Soit la fonction $f(x) = 0.83x^2 + 3.8x + 2.48$. En utilisant le programme précédent, trouvez les solutions pour $f(x) = 0$.

c) La représentation graphique de $f(x)$ est indiquée ci-dessous :



Nous allons utiliser une fonction `EqSecondDegree(a,b,c)` dans **TD N°3** pour reproduire cette figure en utilisant les bibliothèques `numpy` et `matplotlib`.

- Écrivez la fonction `EqSecondDegree(a,b,c)` qui **renvoie** les solutions de l'équation $ax^2 + bx + c = 0$.
- Enregistrez la fonction `EqSecondDegree(a,b,c)` dans un script Python `racines.py`.

d) En utilisant la fonction `EqSecondDegree(a,b,c)`, trouvez les solutions de $f(x) = 0$.

3.6.5 Exercice 5 : programmez une boucle `while`

Définir une séquence de nombres :

$$x_n = n^2 + 1$$

pour les entiers n = 0,1,2,..., N. Écrivez un programme qui affiche x_n pour n = 0,1,..., 20 en utilisant une boucle `while`.

3.6.6 Exercice 6 : Créer une liste avec une boucle `while`

Stockez toutes les valeurs x_n calculées dans l'exercice 5 dans une liste (à l'aide d'une boucle `while`). Afficher la liste complète (en un seul objet).

3.6.7 Exercice 7 : Programmer une boucle `for`

Faites l'exercice 6, mais utilisez une boucle `for`.

3.6.8 Exercice 8 : Écrire une fonction Python

Écrivez une fonction `x(n)` pour calculer un élément dans la séquence $x_n = n^2 + 1$. Appelez la fonction pour n = 4 et afficher le résultat.

3.6.9 Exercice 9 : Renvoyer trois valeurs d'une fonction Python

Écrivez une fonction Python qui évalue les fonctions mathématiques $f(x) = cos(2x)$, $f'(x) = -2sin(2x)$ et $f"(x) = -4cos(2x)$. Retourner ces trois valeurs. Écrivez les résultats de ces valeurs pour $x = \pi$.

Chapitre 4

Bibliothèques scientifiques : numpy et matplotlib

4.1 Bibliothèque numérique : `numpy`

La bibliothèque `numpy` (`http://www.numpy.org`) est utilisée dans presque tous les calculs numériques réalisés à l'aide de Python. C'est une bibliothèque qui fournit des structures de données de haute performance de type vectoriel, matriciel et de dimensions supérieures. Il est implémenté avec les syntaxes de C et Fortran, ainsi lorsque les calculs sont vectorisés les performances sont très bonnes.

Pour utiliser `numpy`, vous devez importer le module :

```
import numpy as np
```

Dans la bibliothèque `numpy`, la terminologie utilisée pour les vecteurs, les matrices et tout ensemble de données à grande dimension est **array**.

4.1.1 Tableaux et matrices

Il existe plusieurs façons d'initialiser de nouveaux tableaux `numpy`, par exemple à partir de :

- Des liste ou des tuples
- Utilisant des fonctions dédiées à la génération de tableaux numpy, tels que `arange`, `linspace`, etc.
- Lecture de données à partir de fichiers

Listes. Par exemple, pour créer de nouveaux tableaux vectoriels et matriciels à partir de listes Python, nous pouvons utiliser la fonction `numpy.array`.

```
In [8]: v = np.array([1,2,3,4]) # Un vecteur: l'argument de la fonction array() est une liste Python
   ...: v
Out[8]: array([1, 2, 3, 4])
```

```
In [9]: M = np.array([[1, 2], [3, 4]]) # Une matrice: l'argument de la fonction de tableau est une liste imbriquée
   ...: M
Out[9]:
array([[1, 2],
       [3, 4]])
```

Les variables `v` et `M` sont de type `ndarray` que fournit le module `numpy`.

```
In [10]: type(v), type(M)
Out[10]: (numpy.ndarray, numpy.ndarray)
```

La différence entre les tableaux `v` et `M` n'est que dans leur forme. Nous pouvons obtenir des informations sur la forme d'un tableau en utilisant la propriété `ndarray.shape`.

```
In [11]: v.shape
Out[11]: (4,)


In [13]: M.shape
Out[13]: (2, 2)
```

Le nombre d'éléments dans le tableau est disponible via la propriété `ndarray.size` :

```
In [14]: M.size
Out[14]: 4
```

De manière équivalente, nous pourrions utiliser la fonction `numpy.shape` et `numpy.size`

```
In [15]: np.shape(M)
Out[15]: (2, 2)
```

et

```
In [16]: np.size(M)
Out[16]: 4
```

Jusqu'à présent, le `numpy.ndarray` ressemble beaucoup à une liste Python (ou à une liste imbriquée).

> **Note:** Pourquoi ne pas utiliser simplement les listes Python pour les calculs au lieu de créer un nouveau type de tableau ?
>
> Il existe plusieurs raisons pour ne pas utiliser que les listes :
> - Les listes de Python sont très générales. Ils peuvent contenir tout type d'objets. Une même liste peut contenir des éléments de différentes natures et changeables. Ils ne prennent pas en charge les fonctions mathématiques telles que les multiplications de matrice et de points, etc. L'implémentation de telles fonctions pour les listes Python ne serait pas très efficace en raison du typage dynamique.
> - Les tableaux Numpy sont **typés** et **homogènes**. Le type d'éléments est déterminé lorsque le tableau est créé.
> - Les tableaux Numpy sont efficaces pour la gestion de la mémoire.
> - En raison du typage statique, la mise en œuvre rapide de fonctions mathématiques telles que la multiplication et l'ajout de tableaux `numpy` peut être implémentée dans

une langue compilée (C et Fortran sont utilisés).

En utilisant la propriété `dtype` (type de données) d'un `ndarray`, on peut connaître le type des éléments d'un tableau ou d'une matrice :

```
In [17]: M.dtype
Out[17]: dtype('int64')
```

Si nous le désirons, nous pouvons définir explicitement le type de données du tableau lorsque nous le créons, ceci en utilisant le mot-clé `dtype` comme argument :

```
In [18]: M_Complex = np.array([[3, 1+2j], [1+2j, 4]], dtype=complex)
    ...: M_Complex
Out[18]: array([[3.+0.j, 1.+2.j],
                [1.+2.j, 4.+0.j]])
```

Note: Les types usuels qui peuvent être utilisés avec `dtype` sont : `int`, `float`, `complex`, `bool`, `object`, etc. Nous pouvons également définir explicitement la taille des bits des éléments, par exemple : `int64`, `int16`, `float128`, `complex128`.

Utilisation de fonctions génératrices de tableaux et de matrices. Pour les tableaux de grandes tailles, il est inconcevable d'initialiser les données manuellement. Au lieu de cela, nous pouvons utiliser l'une des nombreuses fonctions dans `numpy` qui génère des tableaux ou des matrices de différentes formes et tailles. Certains parmi les plus utilisés sont :

- `arange()`
- `linspace()` et `logspace()`
- `mgrid()`
- `diag()`
- `zeros()` et `ones()`
- ...

Fonction `arange()` : Création d'un tableau à l'aide de la fonction `arange()`

```
In [19]: x = np.arange(0, 10, 1) # Arguments: start, stop, step
    ...: x
Out[19]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
In [23]: x = np.arange(-2, 2, 0.1)
    ...: x
Out[23]:
array([-2.00000000e+00, -1.90000000e+00, -1.80000000e+00, -1.70000000e+00,
       -1.60000000e+00, -1.50000000e+00, -1.40000000e+00, -1.30000000e+00,
       -1.20000000e+00, -1.10000000e+00, -1.00000000e+00, -9.00000000e-01,
       -8.00000000e-01, -7.00000000e-01, -6.00000000e-01, -5.00000000e-01,
       -4.00000000e-01, -3.00000000e-01, -2.00000000e-01, -1.00000000e-01,
        1.77635684e-15,  1.00000000e-01,  2.00000000e-01,  3.00000000e-01,
        4.00000000e-01,  5.00000000e-01,  6.00000000e-01,  7.00000000e-01,
        8.00000000e-01,  9.00000000e-01,  1.00000000e+00,  1.10000000e+00,
        1.20000000e+00,  1.30000000e+00,  1.40000000e+00,  1.50000000e+00,
        1.60000000e+00,  1.70000000e+00,  1.80000000e+00,  1.90000000e+00])
```

Fonctions `linspace()` et `logspace()` : En utilisant `linspace()`, les deux points finaux sont inclus

```
In [24]: np.linspace(0, 10, 20) # linspace(star, stop, Nombre de points)
Out[24]:
array([ 0.        ,  0.52631579,  1.05263158,  1.57894737,  2.10526316,
        2.63157895,  3.15789474,  3.68421053,  4.21052632,  4.73684211,
        5.26315789,  5.78947368,  6.31578947,  6.84210526,  7.36842105,
        7.89473684,  8.42105263,  8.94736842,  9.47368421, 10.        ])


In [25]: np.logspace(0, 10, 10, base=e)
Out[25]:
array([1.00000000e+00, 3.03773178e+00, 9.22781435e+00, 2.80316249e+01,
       8.51525577e+01, 2.58670631e+02, 7.85771994e+02, 2.38696456e+03,
       7.25095809e+03, 2.20264658e+04])
```

Fonctions `mgrid()` et `meshgrid()` :

```
In [26]: x, y = np.mgrid[0:5, 0:5] # Similaire à meshgrid dans MATLAB
    ...: x
Out[26]:
array([[0, 0, 0, 0, 0],
       [1, 1, 1, 1, 1],
       [2, 2, 2, 2, 2],
       [3, 3, 3, 3, 3],
       [4, 4, 4, 4, 4]])

In [27]: y
Out[27]:
array([[0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4],
       [0, 1, 2, 3, 4]])


In [31]: x, y = np.meshgrid(np.linspace(0,2,6),np.linspace(0,2,6))
    ...: x
Out[31]:
array([[0. , 0.4, 0.8, 1.2, 1.6, 2. ],
       [0. , 0.4, 0.8, 1.2, 1.6, 2. ],
       [0. , 0.4, 0.8, 1.2, 1.6, 2. ],
       [0. , 0.4, 0.8, 1.2, 1.6, 2. ],
       [0. , 0.4, 0.8, 1.2, 1.6, 2. ],
       [0. , 0.4, 0.8, 1.2, 1.6, 2. ]])

In [32]: y
Out[32]:
array([[0. , 0. , 0. , 0. , 0. , 0. ],
       [0.4, 0.4, 0.4, 0.4, 0.4, 0.4],
       [0.8, 0.8, 0.8, 0.8, 0.8, 0.8],
       [1.2, 1.2, 1.2, 1.2, 1.2, 1.2],
       [1.6, 1.6, 1.6, 1.6, 1.6, 1.6],
       [2. , 2. , 2. , 2. , 2. , 2. ]])
```

Fonction `diag()` : Une matrice diagonale

```
In [40]: np.diag([1,2,3])
Out[40]:
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])
```

Diagonale avec décalage de la diagonale principale

```
In [41]: np.diag([1,2,3], k=3)
Out[41]:
array([[0, 0, 0, 1, 0, 0],
       [0, 0, 0, 0, 2, 0],
       [0, 0, 0, 0, 0, 3],
       [0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0]])
```

Fonctions `zeros()` et `ones()` :

```
In [42]: np.zeros((3,3))
Out[42]:
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

```
In [43]: np.ones((3,3))
Out[43]:
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

4.1.2 Lecture et écriture de données

Écriture de données. Le module `numpy` contient une fonction très pratique, `savetxt`, permettant d'enregistrer des données tabulaires. Les données doivent être stockées dans un tableau numpy à deux dimensions. La fonction `savetxt` permet de contrôler le format des nombres dans chaque colonne (`fmt`), un en-tête peut être ajouté (`header`) et les lignes d'en-tête commencent par un caractère de commentaire (`comment`).

Exemple

Pour stoker les valeurs de $cos(x)$ avec $x \in [0, 2\pi]$ dans un fichier *cosinus.dat*, le code est comme ci-dessous :

```
'''Créer un tableau à deux dimensions de
[x, cos (x)] dans chaque ligne'''
import numpy as np
x = np.linspace(0, 2*np.pi, 200) # 200 valeurs de x
# un tableau 2 colonnes 200 lignes
data = np.array([x, np.cos(x)]).transpose()

# Écrire un tableau de données dans un fichier sous forme de tableau
np.savetxt('cosinus.dat', data, fmt=['%.2f', '%.4f'],
           header='x   cos(x)', comments='#')
```

Le fichier *cosinus.dat* est créé dans le répertoire de travail :

```
# x   cos(x)
0.00 1.0000
0.03 0.9995
0.06 0.9980
0.09 0.9955
0.13 0.9920
```

```
0.16 0.9876
0.19 0.9821
0.22 0.9757
0.25 0.9683
...  ......
...  ......
```

Lecture de données. Les données du fichier *cosinus.dat* peuvent être lues dans un tableau `numpy` par la fonction `loadtxt` (vous pouvez utiliser la commande `help(loadtxt)` dans le notebook pour comprendre l'utilité des arguments entre les parenthèse) :

```
In [51]: data = np.loadtxt('cosinus.dat', comments='#')
    ...: data
Out[51]:
array([[ 0.    ,  1.    ],
       [ 0.03  ,  0.9995],
       [ 0.06  ,  0.998 ],
       [ 0.09  ,  0.9955],
       [ 0.13  ,  0.992 ],
       [ 0.16  ,  0.9876],
       [ 0.19  ,  0.9821],
       [ 0.22  ,  0.9757],
       [ 0.25  ,  0.9683],
       [ 0.28  ,  0.9599],
       ..................
       ..................
       ..................
       [ 6.22  ,  0.998 ],
       [ 6.25  ,  0.9995],
       [ 6.28  ,  1.    ]])
```

> **Note:** Les lignes commençant par le caractère de commentaire sont ignorées lors de la lecture. L'objet `data` résultant est un tableau à deux dimensions : `data[i, j]` contient le numéro de ligne `i` et le numéro de colonne `j` dans la table, c'est-à-dire que `data[i, 0]` contiennent la valeur `x` et que `data[i, 1]` la valeur `cos(x)` dans la i-ème ligne.

Nous pouvons utiliser le fichier *cosinus.dat* pour tracer la fonction $cos(x)$ comme indiqué dans le code suivant :

```
import numpy as np
import matplotlib.pyplot as plt
# Charger les données du fichier 'cosinus.dat'
X, C = np.loadtxt('cosinus.dat', comments='#', unpack=True)
# Tracer C en fonction de X
plt.plot(X,C)
plt.title("cos(x)")
plt.xlabel("X")
plt.ylabel("C")
plt.show()
```

La sortie de ce code est la Figure 4.1.

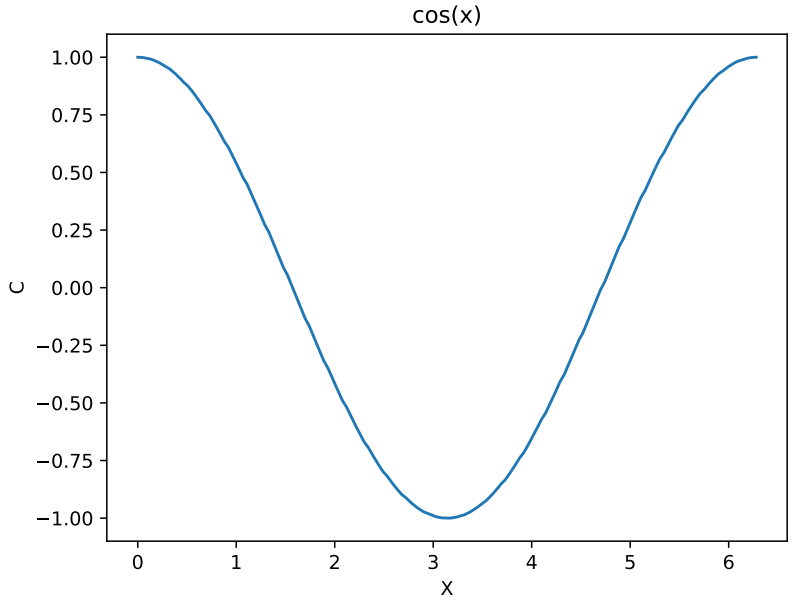


FIGURE 4.1 – Sortie du code.

4.2 Bibliothèque Python de visualisation des données : `matplotlib`

`matplotlib` (`http://matplotlib.org/`) est une excellente bibliothèque graphique 2D et 3D pour générer des graphiques scientifiques. Voici quelques-uns des nombreux avantages de cette bibliothèque :

- Facile à utiliser
- Prise en charge des étiquettes et des textes formatés LaTeX
- Un excellent contrôle des éléments d'une figure, y compris la taille et la résolution (DPI).
- Sortie de haute qualité dans de nombreux formats, y compris PNG, PDF, SVG, EPS, ...
- GUI (Graphical User Interface) pour explorer interactivement les figures.

4.2.1 Documentation en ligne et Galerie

Vous trouverez plus d'informations, y compris une documentation complète avec une vaste galerie d'exemples, sur le site de `mtplotlib`.

De nombreux utilisateurs de `matplotlib` sont souvent confrontés à la question :

> Je veux tracer les courbes de deux fonctions (f te g) **ressemblant** à une troisième (h) ?

Je souhaite bonne chance à ceux qui désirent obtenir rapidement une réponse, même avec l'aide de **google** !. C'est pourquoi la **galerie de matplotlib** (`http://matplotlib.org/gallery.`

`html`) est si utile, car elle montre la variété des possibilités. Ainsi, vous pouvez parcourir la galerie, cliquer sur n'importe quel graphique qui comporte les éléments que vous voulez reproduire et afficher le code qui a servi à le générer. Vous deviendrez rapidement autonome, vous allez mélanger et assortir différents composants pour produire votre propre chef-d'œuvre !

4.2.2 Guide de Démarrage

L'exemple ci-dessous montre comment, de manière très simple, représenter graphiquement la fonction $f(x) = y = x$.

```
# Nom Fichier: BasicPlot1.py
# importaion
import matplotlib.pyplot as plt
# definir x
x = [1, 3, 5, 6, 8, 10, 15]
# definir y
y=x
# créer un nouveau graphique
plt.figure()
# Écrire un titre
plt.title("Figure: f(x) = x")
#plot f(x)= x avec: ligne solide ( - ) et marqueurs ( o ) rouges ( r )
plt.plot(x, y, '-ro')
# Écrire un texte (label) sur l'axe des x
plt.xlabel("X-Axis")
# Écrire un texte (label) sur l'axe des y
plt.ylabel("Y-Axis")
#les graphiques ne seront affichés que lorsque vous appelez plt.show ()
plt.show()
```

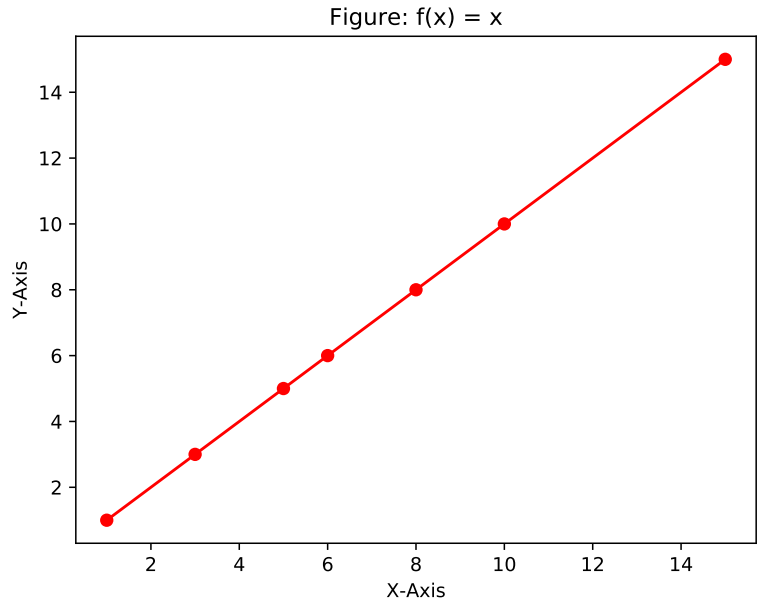


FIGURE 4.2 – Fenêtre de traçage de matplotlib.

Le graphique (`Figure`) est le conteneur de niveau supérieur dans cette hiérarchie. C'est la fenêtre/page globale sur laquelle tout est dessiné. Vous pouvez avoir plusieurs figures indépendantes et les graphiques peuvent contenir plusieurs `Axes`.

La plupart des tracés ont lieu sur des `Axes`. C'est effectivement la zone sur laquelle nous traçons les données et les graduations/labels/etc. qui leur sont associés. Habituellement, nous configurons un `Axes` avec un appel à `Subplot` (qui place les `Axes` sur une grille régulière). Par conséquent, dans la plupart des cas, `Axes` et `Subplot` sont synonymes (figure). Chaque `Axes` ou `Subplot` a un axe X et un axe Y. Ceux-ci contiennent les graduations, les emplacements de graduations, etc.

4.2.3 Vues en grille

Nous avons déjà mentionné qu'une figure peut avoir plus d'un axe. Si vous voulez que vos axes soient sur un système de grille standard, il est alors plus simple d'utiliser `plt.subplot(...)` pour créer un graphique et y ajouter les axes automatiquement.

```
# Nom Fichier: subplots.py
# Importation
import matplotlib.pyplot as plt
fig1=plt.figure(figsize=(5, 5))   # Figure
ax1=plt.subplot(211) # premier subplot dans la figure
ax1.plot([1, 2, 3], "-go")
ax1.set_xlabel("x1")
ax1.set_ylabel("y1")
ax1.set_title('subplot 211') # titre du subplot 211
ax2=plt.subplot(212) # deuxième subplot dans la figure
ax2.plot([4, 2, 5], "--k*")
ax2.set_xlabel("x2")
ax2.set_ylabel("y2")
ax2.set_title('subplot 212') # titre du subplot 212

plt.tight_layout() # Ajuster la figure
plt.show()
```

4.2.4 Commandes de texte de base

Les commandes suivantes permettent de créer du texte dans l'interface `pyplot` :

- `text()` - ajoute du texte à un emplacement quelconque sur les axes ; `matplotlib.axes.Axes.text()`.
- `xlabel()` - ajoute une étiquette à l'axe des x ; `matplotlib.axes.Axes.set_xlabel()`
- `ylabel()` - ajoute une étiquette à l'axe des y ; `matplotlib.axes.Axes.set_ylabel()`
- `title()` - ajoute un titre aux Axes ; `matplotlib.axes.Axes.set_title()`
- `figtext()` - ajoute du texte à un emplacement quelconque sur la figure ; `matplotlib.figure.Figure`
- `suptitle()` - ajoute un titre à la figure ; `matplotlib.figure.Figure.suptitle()`
- `annotate()` - ajoute une annotation, avec une flèche optionnelle, aux axes ; `matplotlib.axes.Axes.a`

Toutes ces fonctions créent et renvoient une instance `matplotlib.text.Text()`, qui peut être configurée avec diverses polices et autres propriétés. L'exemple ci-dessous montre toutes ces commandes en action.

```
# Nom Fichier: BasicText.py
# Importation
import matplotlib.pyplot as plt

fig = plt.figure()
fig.suptitle('Sous-titre de la figure en gras', fontsize=14, fontweight='bold')

ax = fig.add_subplot(111)
```

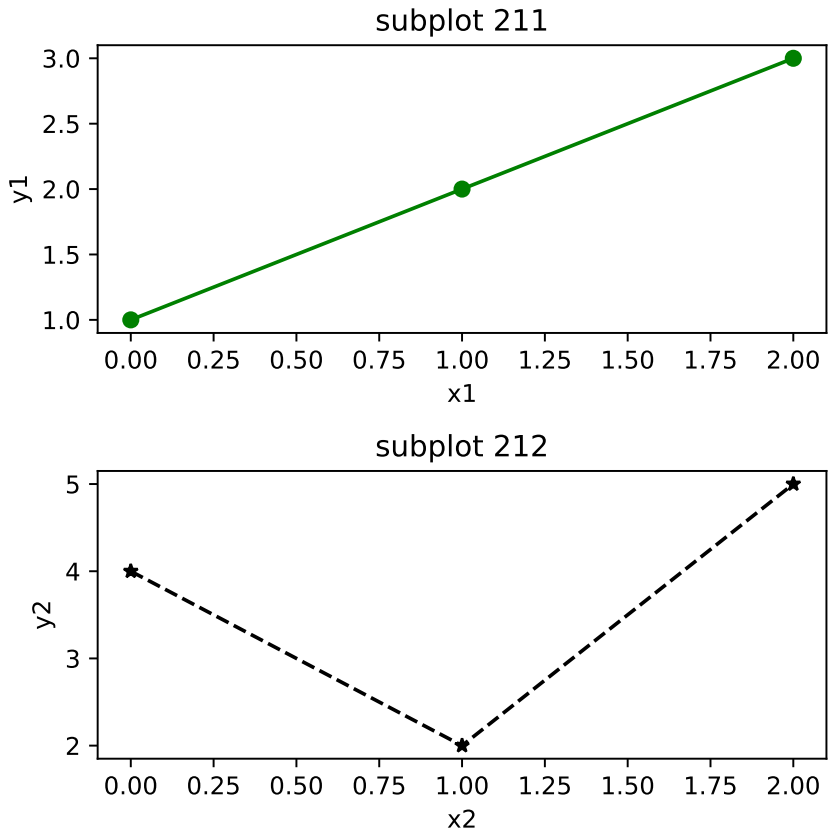


FIGURE 4.3 – Vue en grille, figure(1).

```
fig.subplots_adjust(top=0.85)
ax.set_title('titre des axes')

ax.set_xlabel('xlabel')
ax.set_ylabel('ylabel')

ax.text(3, 8, 'texte en italique encadré par les données', style='italic',
        bbox={'facecolor':'blue', 'alpha':0.5, 'pad':10})

ax.text(2, 6, r'Une équation: $E=mc^2$', fontsize=15)


ax.text(0.95, 0.01, "texte en couleur sur l'axe des x",
        verticalalignment='bottom', horizontalalignment='right',
        transform=ax.transAxes,
        color='red', fontsize=14)

ax.plot([2], [1], 'o')

ax.annotate('annotation: point M(2,1)', xy=(2, 1), xytext=(3, 4),
            arrowprops=dict(facecolor='black', shrink=0.05))

ax.axis([0, 10, 0, 10])
plt.grid()
plt.show()
```

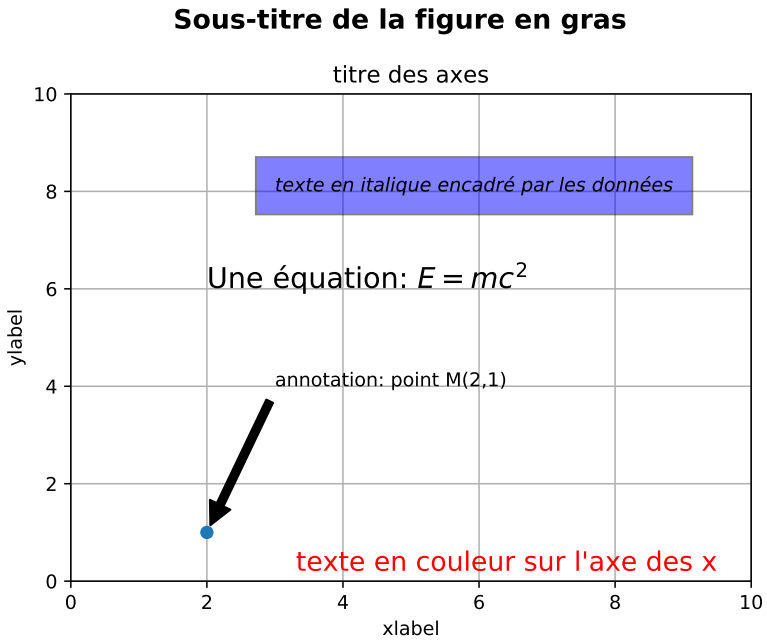


FIGURE 4.4 – Texte de base.

4.2.5 Styles de lignes et de marqueurs

Pour changer la largeur de ligne, nous pouvons utiliser l'argument de mot-clé `linewidth` ou `lw`, et le style de ligne peut être sélectionné à l'aide des arguments de mot-clé `linestyle` ou `ls` :

```
# Nom Fichier: LineandMarkerStyles.py
# Importation
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0, 5, 10)
fig, ax = plt.subplots()

ax.plot(x, x+1, color="blue", linewidth=0.25)
ax.plot(x, x+2, color="blue", linewidth=0.50)
ax.plot(x, x+3, color="blue", linewidth=1.00)
ax.plot(x, x+4, color="blue", linewidth=2.00)

# possible linestype options '-', '-.', ':', 'steps'
ax.plot(x, x+5, color="red", lw=2, linestyle='-')
ax.plot(x, x+6, color="red", lw=2, ls='-.')
ax.plot(x, x+7, color="red", lw=2, ls=':')

# custom dash
line, = ax.plot(x, x+8, color="black", lw=1.50)
line.set_dashes([5, 10, 15, 10]) # format: line length, space length, ...

# possible marker symbols: marker = '+', 'o', '*', 's', ',', '.', '1', '2', '3', '4', ...
ax.plot(x, x+ 9, color="green", lw=2, ls='-.', marker='+')
ax.plot(x, x+10, color="green", lw=2, ls='-.', marker='o')
ax.plot(x, x+11, color="green", lw=2, ls='-.', marker='s')
ax.plot(x, x+12, color="green", lw=2, ls='-.', marker='1')

# marker size and color
ax.plot(x, x+13, color="purple", lw=1, ls='-', marker='o', markersize=2)
ax.plot(x, x+14, color="purple", lw=1, ls='-', marker='o', markersize=4)
ax.plot(x, x+15, color="purple", lw=1, ls='-', marker='o', markersize=8, markerfacecolor="red")
ax.plot(x, x+16, color="purple", lw=1, ls='-', marker='s', markersize=8,
```

```
        markerfacecolor="yellow", markeredgewidth=2, markeredgecolor="blue")
# make a title for the subplot
ax.set_title('"ax.plot(x, y, ...)": Lines and/or markers', fontsize=16, weight='bold')
# make x and y axis label and set their font size and weight
ax.set_xlabel("X-Axis", fontsize=12, weight='bold')
ax.set_ylabel("Y-Axis", fontsize=12, weight='bold')
plt.show()
```

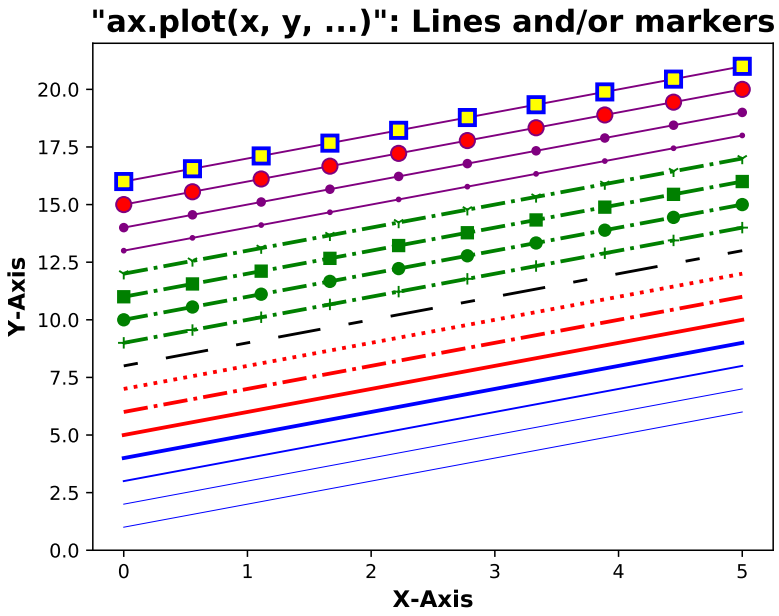


FIGURE 4.5 – Styles de lignes et de marqueurs.

4.2.6 Colormap : Tracés contour, Imshow et 3D

Voir la documentation de matplotlib colormaps `http://matplotlib.org/users/colormaps.html`.

Tracés contour.

```
# Nom Fichier: ContourPlot.py
# Importation
import numpy as np
import matplotlib.pyplot as plt

def f(x,y):
    return (1 - x / 2 + x**5 + y**3) * np.exp(-x**2 -y**2)

n = 256
x = np.linspace(-3, 3, n)
y = np.linspace(-3, 3, n)
X,Y = np.meshgrid(x, y)

# Surface
S = plt.contourf(X, Y, f(X, Y), 8, cmap='plasma')
# Contour
C = plt.contour(X, Y, f(X, Y), 8, colors='black')
```

```
plt.clabel(C, inline=1, fontsize=10)

plt.colorbar(S) # afficher la barre de couleurs
plt.show()
```

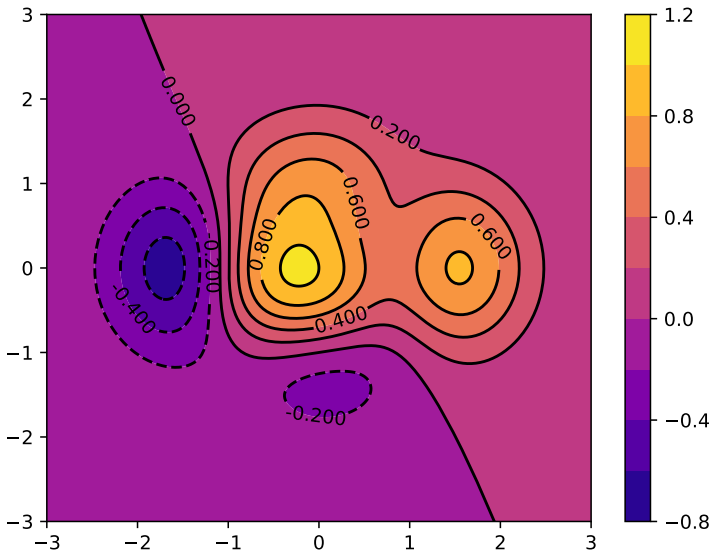


FIGURE 4.6 – Exemple de tracé de contour.

Imshow (Image pixelisée).

```
# Nom Fichier: Imshow.py
# Importation
import numpy as np
import matplotlib.pyplot as plt

def f(x, y):
    return (1 - x / 2 + x ** 5 + y ** 3 ) * np.exp(-x ** 2 - y ** 2)

n = 30
x = np.linspace(-3, 3, n)
y = np.linspace(-3, 3, n)
X, Y = np.meshgrid(x, y)
Z = f(X, Y)
# Image
im = plt.imshow(Z, interpolation='nearest', cmap='gray', origin='lower')

plt.colorbar(im)
plt.show()
```

Tracé en 3D.

```
# Nom Fichier: Plot3D.py
# Importation
import numpy as np
import matplotlib.pyplot as plt
```

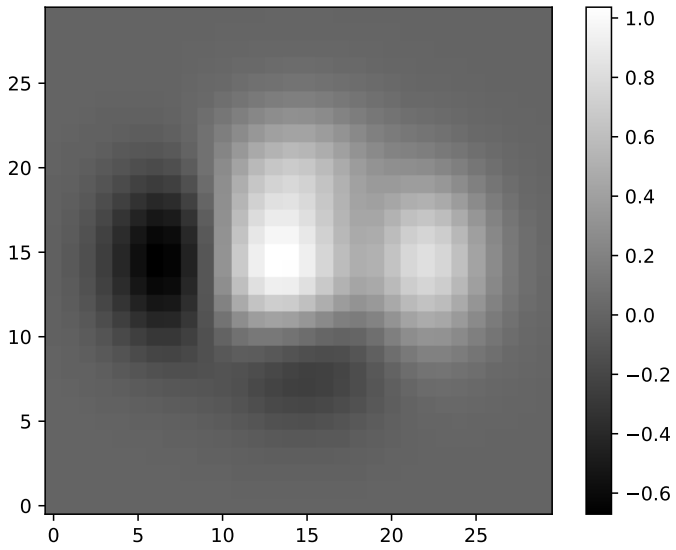


FIGURE 4.7 – Exemple d'image pixelisée.

```
from mpl_toolkits.mplot3d import Axes3D
def f(x, y):
    return (1 - x / 2 + x ** 5 + y ** 3 ) * np.exp(-x ** 2 - y ** 2)
n = 100
x = np.linspace(-3, 3, n)
y = np.linspace(-3, 3, n)
X, Y = np.meshgrid(x, y)
Z = f(X,Y)
fig = plt.figure()
ax = Axes3D(fig)
S = ax.plot_surface(X, Y, Z, cmap="viridis")
ax.contourf(X, Y, Z, zdir='z', offset=-1, cmap="viridis")
ax.set_zlim(-1, 1)
plt.colorbar(S, shrink=0.5)
plt.show()
```

4.3 Travaux dirigés

4.3.1 Exercice 1 : Tracer une fonction

Écrire un programme qui trace la fonction $g(y) = e^{-y}sin(4y)$ pour $y \in [0,4]$ en utilisant une ligne continue rouge. Utilisez 500 intervalles pour évaluer les points dans [0,4]. Stockez toutes les coordonnées et les valeurs dans des tableaux. Placez le texte des graduations sur les axes et utilisez le titre "Onde sinusoïdale atténuée".

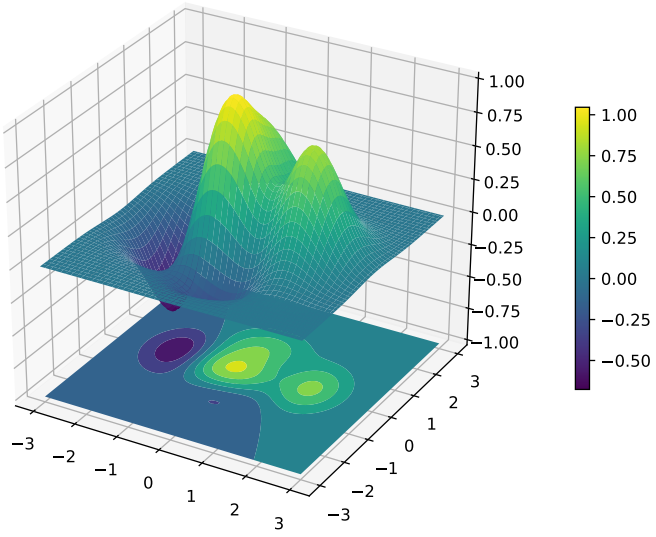


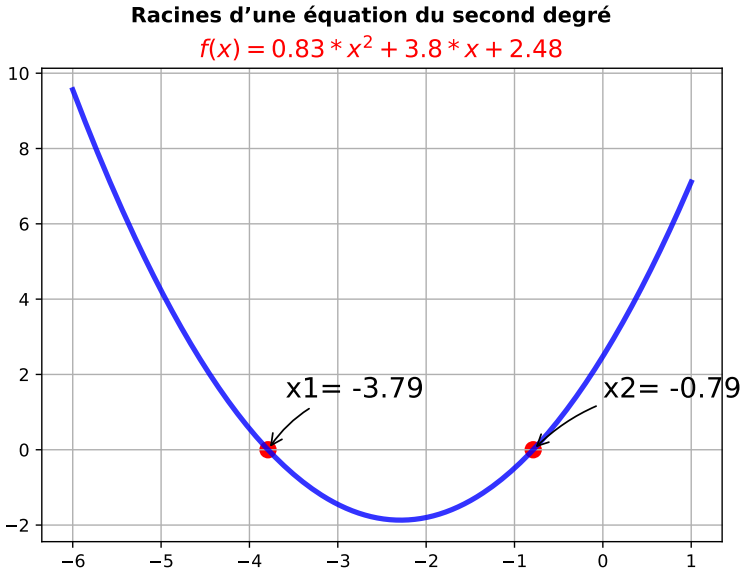
FIGURE 4.8 – Exemple de tracé en 3D.

4.3.2 Exercice 2 : Tracer deux fonctions

Comme Exercice 1, mais ajouter une courbe en pointillé noir pour la fonction $h(y) = e^{-\frac{3}{2}y} sin(4y)$. Inclure une légende pour chaque courbe (avec les noms g et h).

4.3.3 Exercice 3 : Racines d'une équation du second degré

Dans l'"application de l'exercice 4 dans TD N°2, nous avons montré la représentation graphique d'une équation du second degré $f(x) = 0.83x^2 + 3.8x + 2.48$ ainsi que ses racines réelles :



Reproduire ce graphique en utilisant la fonction `EqSecondDegree(a,b,c)` du script Python `racines.py` pour déterminer les valeurs des racines x1 et x2 de l'équation $f(x)$.

4.3.4 Exercice 4 : Approximer une fonction par une somme de sinus

Nous considérons la fonction constante par morceaux :

$$f(t) = \begin{cases} 1, & 0 < t < T/2, \\ 0, & t = T/2, \\ -1, & T/2 < t \leq T \end{cases} \tag{4.1}$$

On peut approcher f(t) par la somme :

$$S(t;n) = \frac{4}{\pi} \sum_{i=1}^{n} \frac{1}{2i-1} \sin\left(\frac{2(2i-1)\pi t}{T}\right) \tag{4.2}$$

On peut montrer que $S(t;n) \to f(t)$ quand $n \to \infty$

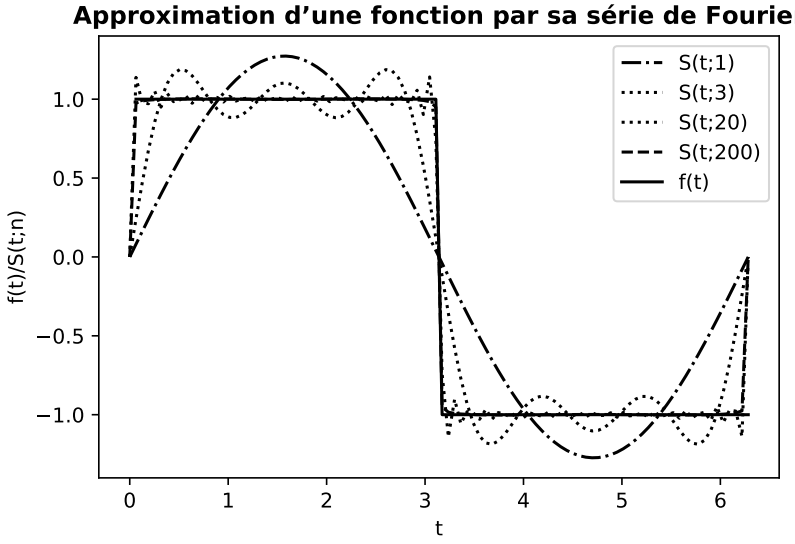
a) Ecrivez une fonction Python `S(t, n, T)` pour renvoyer la valeur de $S(t;n)$.

b) Ecrivez une fonction Python `f(t, T)` pour calculer $f(t)$.

c) Créer un tableau `t` à l'aide de la fonction `linspace`, du module `numpy`, pour `100` valeurs `t` uniformément espacés dans [0, T]. On prendra $T = 2\pi$.

d) Remplir une liste `F` par les valeurs de `f(ti,T)` avec $ti \in t$. Transformer la liste `F` en un tableau (nous voulons avoir un tableau pour la fonction $f(t)$ avec $t \in [0,T]$ et $T = 2\pi$).

e) Tracer $S(t;1)$, $S(t;3)$, $S(t;20)$, $S(t;200)$ et la fonction exacte $f(t)$ dans le même graphique. Le résultat devrait être similaire au graphique ci-dessous.



f) Quelle est la relation entre la qualité de l'approximation et le choix de la valeur de `n` ?

4.3.5 Exercice 5 : Fonctions spéciales (intégrales de Fresnel et spirale de Cornu)

Les intégrales de Fresnel ont été introduites par le physicien français Augustin Fresnel (1788-1827) lors de ses travaux sur les interférences lumineuses (voici un article intéressant à lire : Fresnel, des Mathématiques en Lumière).

Ces intégrales doivent être calculées numériquement à partir des développements en série des intégrales :

$$\int_0^x e^{-i\frac{\pi t^2}{2}}dt = \int_0^x cos(t^2)dt - i\int_0^x sin(t^2)dt = C(x) - iS(x)$$

Les fonctions de Fresnel sont des fonctions spéciales, définies par :

Pour $x \geq \sqrt{\frac{8}{\pi}}$

$$C(x) = \frac{1}{2} + \cos\left(\frac{\pi x^2}{2}\right) gg1 + \sin\left(\frac{\pi x^2}{2}\right) ff1$$
$$S(x) = \frac{1}{2} - \cos\left(\frac{\pi x^2}{2}\right) ff1 + \sin\left(\frac{\pi x^2}{2}\right) gg1$$

et pour $0 \leq x < \sqrt{\frac{8}{\pi}}$

$$C(x) = \cos\left(\frac{\pi x^2}{2}\right) gg2 + \sin\left(\frac{\pi x^2}{2}\right) ff2$$
$$S(x) = -\cos\left(\frac{\pi x^2}{2}\right) ff2 + \sin\left(\frac{\pi x^2}{2}\right) gg2$$

Où :

$$ff1 = \sum_{n=0}^{11} \frac{d_n}{x^{2n+1}} \left(\frac{8}{\pi}\right)^{n+1/2} \quad gg1 = \sum_{n=0}^{11} \frac{c_n}{x^{2n+1}} \left(\frac{8}{\pi}\right)^{n+1/2}$$
$$ff2 = \sum_{n=0}^{11} b_n x^{2n+1} \left(\frac{\pi}{8}\right)^{n+1/2} \quad gg2 = \sum_{n=0}^{11} a_n x^{2n+1} \left(\frac{\pi}{8}\right)^{n+1/2}$$

et a_n, b_n, c_n et d_n sont des coefficients tabulés (*J.Boersma Math Computation 14,380(1960)*) et donnés dans un fichier **coef.dat** :

```
#-------------------------------------------------
#    an            bn           cn           dn
#-------------------------------------------------
+1.595769140 -0.000000033 -0.000000000 +0.199471140
-0.000001702 +4.255387524 -0.024933975 +0.000000023
-6.808568854 -0.000092810 +0.000003936 -0.009351341
-0.000576361 -7.780020400 +0.005770956 +0.000023006
+6.920691902 -0.009520895 +0.000689892 +0.004851466
-0.016898657 +5.075161298 -0.009497136 +0.001903218
-3.050485660 -0.138341947 +0.011948809 -0.017122914
-0.075752419 -1.363729124 -0.006748873 +0.029064067
+0.850663781 -0.403349276 +0.000246420 -0.027928955
-0.025639041 +0.702222016 +0.002102967 +0.016497308
-0.150230960 -0.216195929 -0.001217930 -0.005598515
+0.034404779 +0.019547031 +0.000233939 +0.000838386
```

Écrire un programme Python qui calcule les fonctions de Fresnel $C(x)$ et $S(x)$ ainsi que leurs représentations graphiques :

a) Définir les fonctions `ff1(x)`, `gg1(x)`, `ff2(x)` et `gg2(x)`. Chaque fonction renvoie la valeur de la somme qui lui correspond.

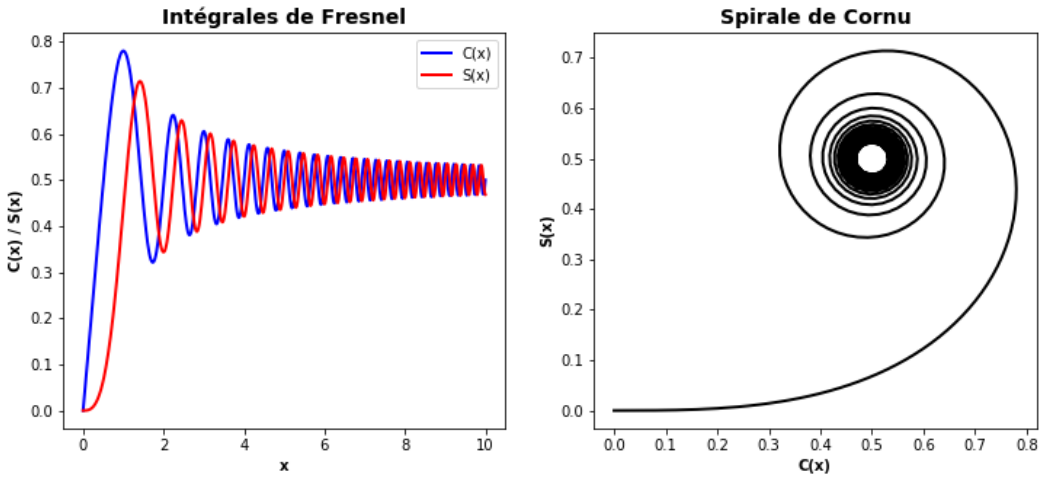
b) Définir les fonctions Python `C(x)` et `S(x)` qui renvoient respectivement les listes, les valeurs de $C(x)$ et $S(x)$, `CF` et‘ SF‘ (en utilisant une boucle `for` pour remplir les listes par exemple).

c) Créer des tableaux `an`, `bn`, `cn` et `dn` à partir du fichier **coef.dat**.

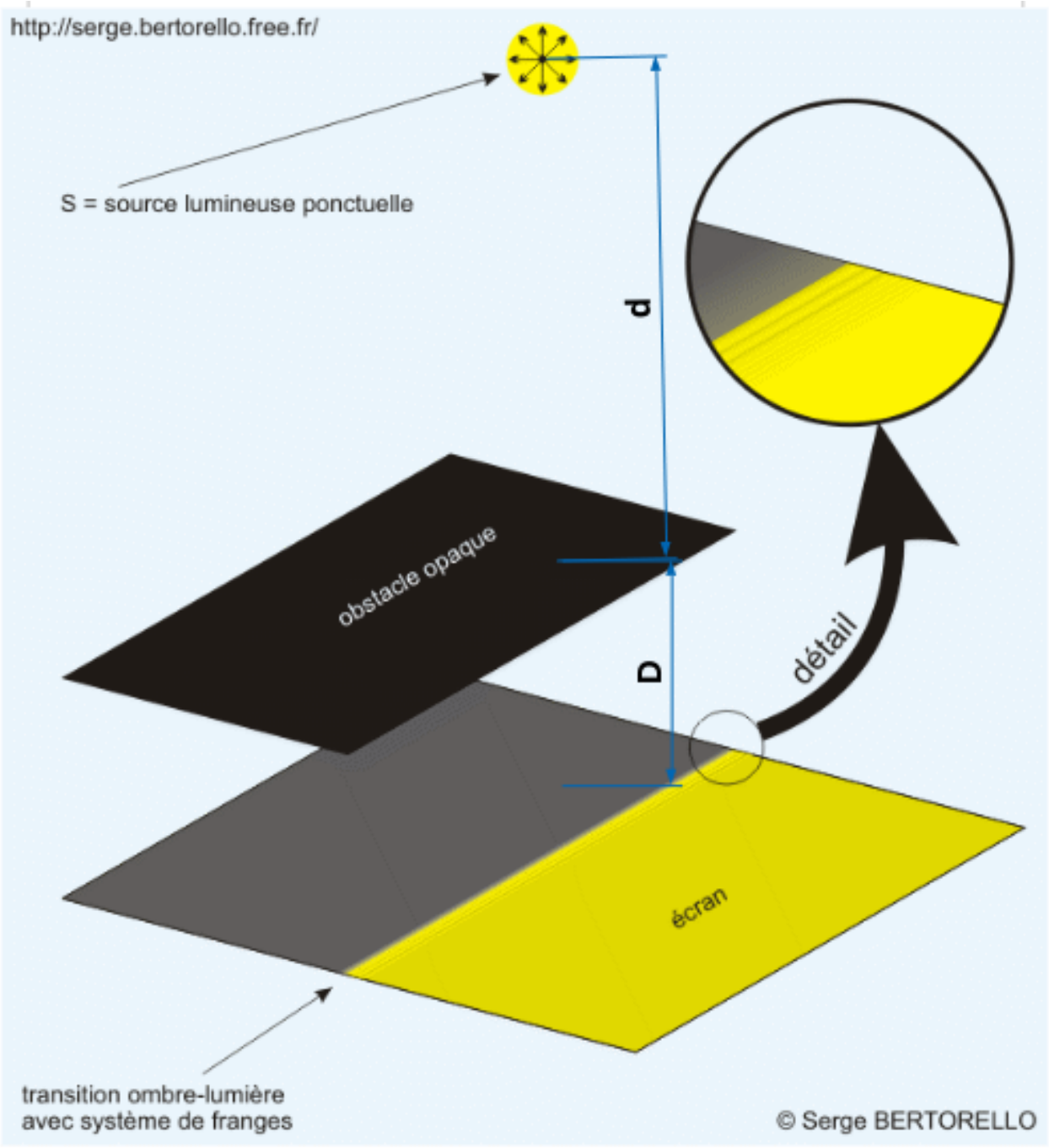
d) Créer un tableau `x`. Utilisez 800 intervalles pour évaluer les points dans [0,10] (cas où $x \geq 0$).

e) Transformer `C(x)` et `S(x)` en tableaux `numpy`, respectivement `CF` et `SF`.

f) Tracer une grille de figures à 2 colonnes (voir Cours3 : Vues en grille) dont le graphique de gauche représente `CF` et `SF` en fonction de `x` et le graphique de droite représente une clothoïde (ou spirale de Cornu, ou Spirale de Fresnel..)‘SF‘ en fonction de `CF`. La sortie de ce programme devrait être comme suit :



g) Application : Simulation de diffraction par un bord



On reprend cette fois les expressions obtenues dans le traitement du bord avec la spiral de Cornu. On considère que la source est à l'infini, soit d→∞. L'expression (1.19) devient :

$$u = \sqrt{\frac{2d}{D(d+D)\lambda}}\, x^M = \sqrt{\frac{2}{D\lambda}}\, x^M$$

Un bord peut être considéré comme un rectangle de largeur infinie et de hauteur également infinie sur lequel la lumière ne rencontre qu'un seul de ses bord au milieu de sa hauteur. Cela revient à faire tendre $v_1 \to \infty$ et $v_2 \to \infty$, ainsi que $u_1 \to \infty$ (ou $u2 \to \infty$). Dans ce cas, l'expression de l'intensité (1.15) se simplifie énormément :

$$I = \frac{I_0}{4}\left[(C(\infty) - C(u))^2 + (S(\infty) - S(u))^2\right].\left[(C(\infty) - C(\infty))^2 + (S(\infty) - S(\infty))^2\right]$$

$$I = \frac{I_0}{2}\left[(0.5 - C(u))^2 + (0.5 - S(u))^2\right]$$

Écrire un programme Python que a pour ojectif de reproduire la figure cui-dessous et qui représente les variations de I pour une lumière de $\lambda = 592$ nm et $D = 1$ m du bord.

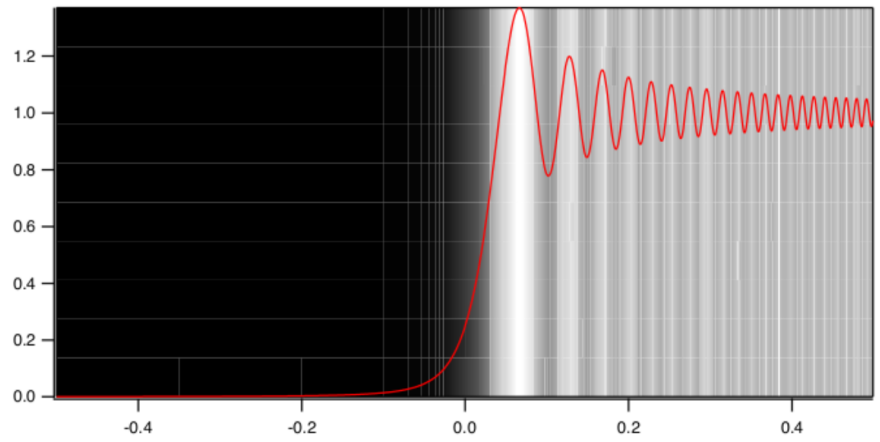


FIGURE 4.9 – Diffraction par un bord pour $\lambda = 592$ nm et D = 1 m.

4.4 TP : Diagramme de rayonnement

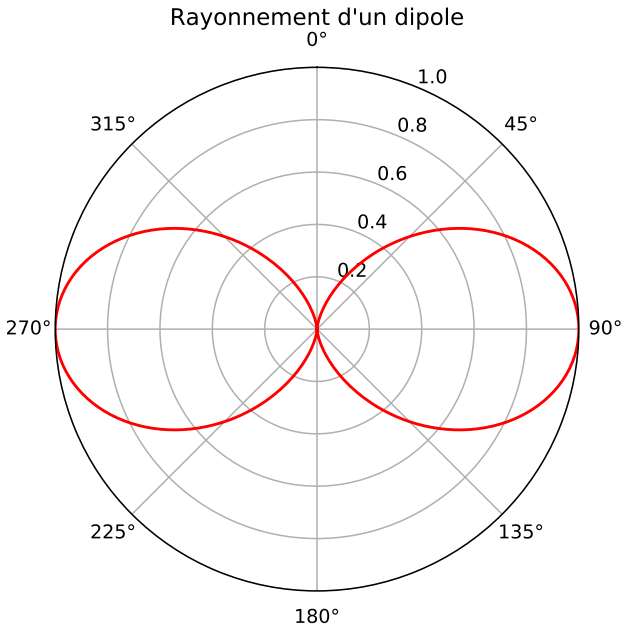
La puissance surfacique rayonnée dépend localement de l'angle θ. Elle est maximale dans le plan équatorial et s'annule dans l'axe du dipôle. Pour représenter graphiquement cette dépendance angulaire, on trace la courbe suivante en coordonnées polaires :

$$\rho(\theta) = sin^2\theta \tag{4.3}$$

4.4.1 Simulation Python

```
## NOM DU PROGRAMME: DipRadiation.py
#% IMPORTATION
```

```
import numpy as np
import matplotlib.pyplot as plt

theta = np.linspace(0,2*np.pi,500)
rho = np.sin(theta)**2
plt.figure()
ax = plt.subplot(111, polar=True)
ax.plot(theta, rho,color='r')
ax.set_theta_zero_location('N')
ax.set_theta_direction(-1)
ax.set_rmax(1.0)
ax.set_title("Rayonnement d'un dipole", va='bottom')
theta = np.linspace(0.01,2*np.pi,500)
plt.tight_layout()
plt.savefig("dipole1.png"); plt.savefig("dipole1.pdf")
plt.show()
```

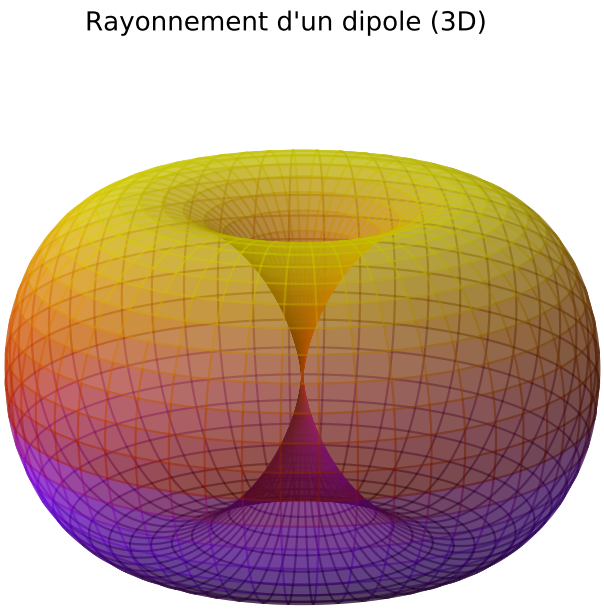


Ce tracé est un diagramme de rayonnement (ou indicatrice de rayonnement). Il permet de voir comment évolue la puissance avec l'angle. On voit par exemple que la puissance est égale à la moitié de sa valeur maximale pour un angle de 45 degrés.

Pour tracer le diagramme de rayonnement sur un graphe 3D, nous devons passer par une fonction de conversion de coordonnées sphériques/cartésiennes ; `sph2cart(azimuth,elevation,r)` :

```
## NOM DU PROGRAMME: DipRadiation3D.py
#% IMPORTATION
import numpy as np
import matplotlib.pyplot as plt
import mpl_toolkits.mplot3d.axes3d as axes3d
import matplotlib.colors as mcolors

def sph2cart(azimuth,elevation,r):
    """
    Convertisseur de Coordonnée Sphérique/Cartésienne
    """
    x = r * np.sin(elevation) * np.cos(azimuth)
    y = r * np.sin(elevation) * np.sin(azimuth)
```

```
    z = r * np.cos(elevation)
    return x, y, z
theta = np.linspace(0.001,np.pi,400)
phi = np.linspace(0.001,2*np.pi,400)
THETA, PHI = np.meshgrid(theta,phi)
RHO = np.sin(THETA)**2
X, Y, Z = sph2cart(PHI,THETA,RHO)
fig = plt.figure(figsize=(8,5))
cmap = plt.get_cmap('gnuplot')
ax2 = plt.subplot(111, projection='3d')
ax2._axis3don = False # hide x, y, z axis
norm = mcolors.Normalize(vmin=Z.min(), vmax=Z.max())
ax2.plot_surface(X, Y, Z, rstride=8, cstride=8,
                 facecolors=cmap(norm(Z)), antialiased=True, alpha=0.5)
fig.suptitle("Rayonnement d'un dipole (3D)")
plt.tight_layout()
plt.savefig("dipole1_3D.png"); plt.savefig("dipole1_3D.pdf")
plt.show()
```



4.4.2 Antenne dipolaire

Pour une antenne dipolaire de longueur L, le calcul est très complexe car on ne connaît pas a priori l'expression de l'intensité I(z,t) du courant dans l'antenne. Les calculs conduisent à l'expression suivante :

$$I(z,t) = I_0 sin(\frac{L}{2} - z)e^{-i\omega t} \tag{4.4}$$

On par ailleurs I(-z,t)=I(z,t). Le courant s'annule aux extrémités et varie sinusoïdalement avec une période λ. Connaissant le courant, on peut calculer le champ électromagnétique en

sommant les contributions des segments élémentaires, qui sont des dipôles oscillants.

Pour une antenne de longueur petite devant la longueur d'onde, le courant décroît linéairement entre sa valeur I0 au centre de l'antenne et une valeur nulle à l'extrémité. Dans ce cas, on peut utiliser les résultats du dipôle oscillant en remplaçant I0 par I0/2.

Il est intéressant d'augmenter la longueur des antennes car la puissance émise est proportionnelle au carré de la longueur. Pour des fréquences supérieures à 100 MHz, on utilise des antennes dont la longueur n'est pas petite devant la longueur d'onde. Par exemple, une antenne demi-onde a une longueur égale à $\lambda/2$. Dans le cas général, le facteur angulaire du champ électrique est la fonction suivante :

$$f(\theta) = \frac{cos(\frac{kL}{2}cos\theta) - cos(\frac{kL}{2})}{sin\theta} \tag{4.5}$$

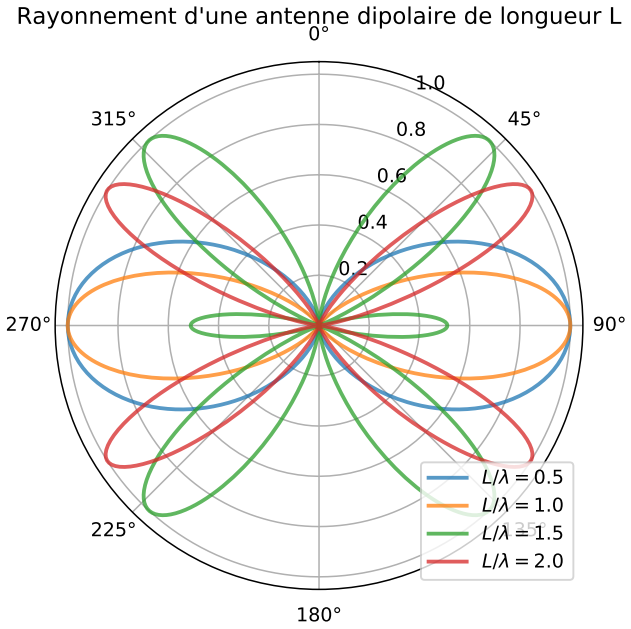
Le carré de cette fonction permet de tracer le diagramme de rayonnement en fonction du rapport

$$\frac{kL}{2} = \frac{\pi L}{\lambda} \tag{4.6}$$

4.4.3 Simulation Python

```
## NOM DU PROGRAMME: DipRadiationGeneral.py
#% IMPORTATION
import numpy as np
import matplotlib.pyplot as plt
import mpl_toolkits.mplot3d.axes3d as axes3d

def rho(theta, rapport = 0.5):
    u = rapport* np.pi
    F = (np.cos(u*np.cos(theta)) - np.cos(u))/(np.sin(theta))
    G = F * F
    return G/G.max()

plt.figure()
ax = plt.subplot(111, polar=True)
plt.title("Rayonnement d'une antenne dipolaire de longueur L")
ax.set_theta_zero_location('N')
ax.set_theta_direction(-1)
ax.set_rmax(1.0)
theta = np.linspace(0.01,2*np.pi,500)

for rapport in [0.5,1.0,1.5, 2]:
    ax.plot(theta,rho(theta, rapport), lw = 2,
            label=r"$L/\lambda=%.1f$"%rapport, alpha=0.75)
plt.legend(loc='lower right')
plt.tight_layout()
plt.savefig("dipole.png"); plt.savefig("dipole.pdf")
plt.show()

#% Dipôle 3D
Theta = np.linspace(0.001,np.pi,400)
Phi = np.linspace(0.001,2*np.pi,400)
THETA, PHI = np.meshgrid(Theta,Phi)
#
def sph2cart(azimuth,elevation,r):
    """
    Convertisseur de Coordonnée Sphérique/Cartésienne
    """
    x = r * np.sin(elevation) * np.cos(azimuth)
    y = r * np.sin(elevation) * np.sin(azimuth)
    z = r * np.cos(elevation)
    return x, y, z
```

```
import matplotlib.colors as mcolors
fig = plt.figure(figsize=(8,5))
cmap = plt.get_cmap('gnuplot')
rapport = 1.4
X, Y, Z = sph2cart(PHI,THETA,rho(THETA,rapport))
#ax1 = plt.subplot(121, polar=True)
#ax1.plot(theta,rho(theta, rapport), lw = 2)
#ax1.set_theta_zero_location('N')
#ax1.set_theta_direction(-1)
ax2 = plt.subplot(111, projection='3d')
ax2._axis3don = False # hide x, y, z axis
norm = mcolors.Normalize(vmin=Z.min(), vmax=Z.max())
ax2.plot_surface(X, Y, Z, rstride=8, cstride=8,
                 facecolors=cmap(norm(Z)), antialiased=True, alpha=0.5)
fig.suptitle("Diagramme de rayonnement: "+r"$L/\lambda=%.1f$"%rapport)
plt.tight_layout()
plt.savefig("dipole3D_w14.png"); plt.savefig("dipole3D_w14.pdf")
```



On voit que la directivité de l'émission augmente avec la longueur de l'antenne. Pour $L > \lambda$, il apparaît des lobes à 45 degrés, qui deviennent prépondérants lorsque la longueur augmente.

Diagramme de rayonnement: $L/\lambda = 0.5$

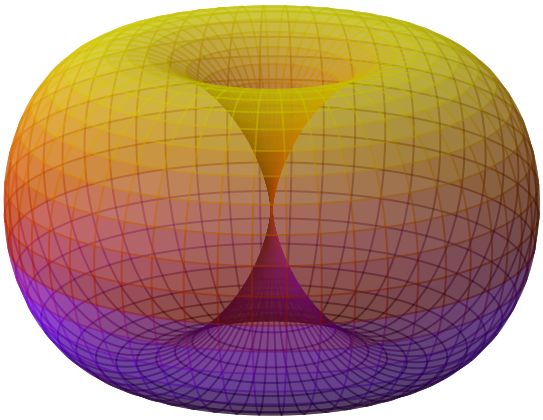


Diagramme de rayonnement: $L/\lambda = 1.4$

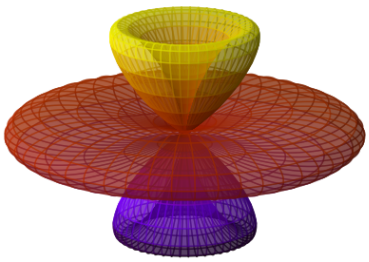


Diagramme de rayonnement: $L/\lambda = 1.5$

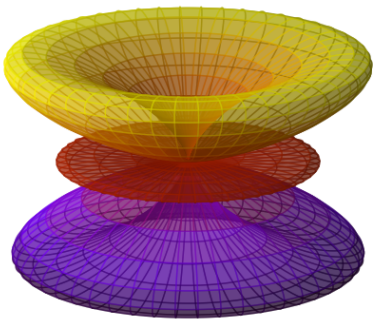


Diagramme de rayonnement: $L/\lambda = 2.0$

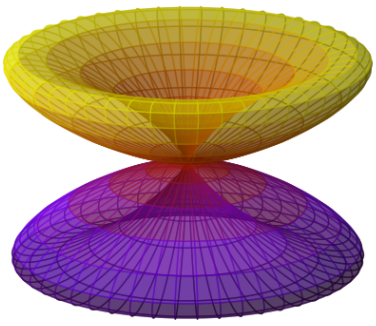
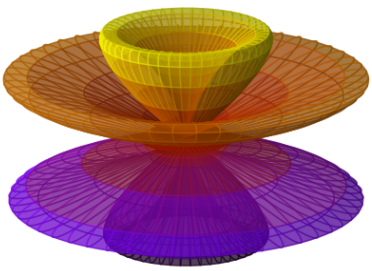


Diagramme de rayonnement: $L/\lambda = 2.4$



Chapitre 5

Intégration numérique

5.1 Introduction

L'intégration numérique est un chapitre important de l'analyse numérique et un outil indispensable en physique numérique. On intègre numériquement dans deux cas principaux :

- on ne peut pas intégrer analytiquement,
- l'intégrande est fourni non pas sous la forme d'une fonction mais de tableaux de mesures, cas d'ailleurs le plus fréquent dans la vraie vie.

Les méthodes numériques d'intégration d'une fonction sont nombreuses et les techniques très diverses. Des très simples, comme la méthode des rectangles aux très complexes comme certaines variétés de la méthode de Monte-Carlo.

5.2 Idées de base de l'intégration numérique

Nous considérons l'intégrale

$$\int_a^b f(x)dx \tag{5.1}$$

La plupart des méthodes numériques de calcul de cette intégrale divisent l'intégrale d'origine en une somme de plusieurs intégrales, chacune couvrant une partie plus petite de l'intervalle d'intégration d'origine $[a,b]$. Cette réécriture de l'intégrale est basée sur une sélection de points d'intégration x_i, $i = 0,1,\ldots,n$ qui sont répartis sur l'intervalle $[a,b]$. Les points d'intégration peuvent ou non être répartis uniformément. Une distribution uniforme simplifie les expressions et est souvent suffisante, nous nous limiterons donc principalement à ce choix. Les points d'intégration sont ensuite calculés comme :

$$x_i = a + ih, \quad i = 0,1,\ldots,n \tag{5.2}$$

où

$$h = \frac{b-a}{n} \tag{5.3}$$

Compte tenu des points d'intégration, l'intégrale d'origine est réécrite sous la forme d'une somme d'intégrales, chaque intégrale étant calculée sur le sous-intervalle entre deux points d'intégration consécutifs. L'intégrale dans (5.1) est donc exprimée comme :

$$\int_a^b f(x)dx = \int_{x_0}^{x_1} f(x)dx + \int_{x_1}^{x_2} f(x)dx + \ldots + \int_{x_{n-1}}^{x_n} f(x)dx \tag{5.4}$$

Notez que $x_0 = a$ et $x_n = b$.

En partant de (5.4), les différentes méthodes d'intégration différeront dans la façon dont elles approchent chaque intégrale du côté droit. L'idée fondamentale est que chaque terme est une intégrale sur un petit intervalle $[x_i, x_{i+1}]$, et sur ce petit intervalle, il est logique d'approximer f par une forme simple, disons une constante, une ligne droite ou une parabole , que nous pouvons facilement intégrer à la main. Les détails deviendront clairs dans les exemples à venir.

5.2.1 Exemple de calcul

Pour comprendre et comparer les méthodes d'intégration numérique, il est avantageux d'utiliser une intégrale spécifique pour les calculs et les illustrations graphiques. En particulier, nous voulons utiliser une intégrale que nous pouvons calculer à la main de sorte que la précision des méthodes d'approximation puisse être facilement évaluée. Notre intégrale spécifique est tirée de la physique de base. Supposons que vous accélérez votre voiture du repos et demandez-vous jusqu'où vous allez en T secondes. La distance est donnée par l'intégrale $\int_0^T v(t)dt$, où $v(t)$ est la vitesse en fonction du temps. Une fonction de vitesse en augmentation rapide pourrait être :

$$v\,(t) = 3t^2 e^{t^3} \tag{5.5}$$

La distance après une seconde est

$$\int_0^1 v(t)dt \tag{5.6}$$

qui est l'intégrale que nous cherchons à calculer par des méthodes numériques. Heureusement, l'expression choisie de la vitesse a une forme qui permet de calculer facilement la primitive comme

$$V(t) = e^{t^3} - 1 \tag{5.7}$$

Nous pouvons donc calculer la valeur exacte de l'intégrale comme $V(1) - V(0) \approx 1.718$ (arrondi à 3 décimales pour plus de commodité).

5.3 La règle du trapèze composite

L'intégrale $\int_a^b f(x)dx$ peut être interprété comme l'aire entre l'axe des x et le graphique $y = f(x)$ de fonction à intégrer. La figure 5.1 illustre cette zone de choix (5.6). Le calcul de l'intégrale $\int_0^1 f(t)dt$ revient à calculer l'aire de la zone hachurée.

Si nous remplaçons le vrai graphique de la figure 5.1 par un ensemble de segments de ligne droite, nous pouvons voir la zone plutôt comme composée de trapèzes, dont les zones sont faciles à calculer. Ceci est illustré sur la figure 5.2, où 4 segments de ligne droite donnent

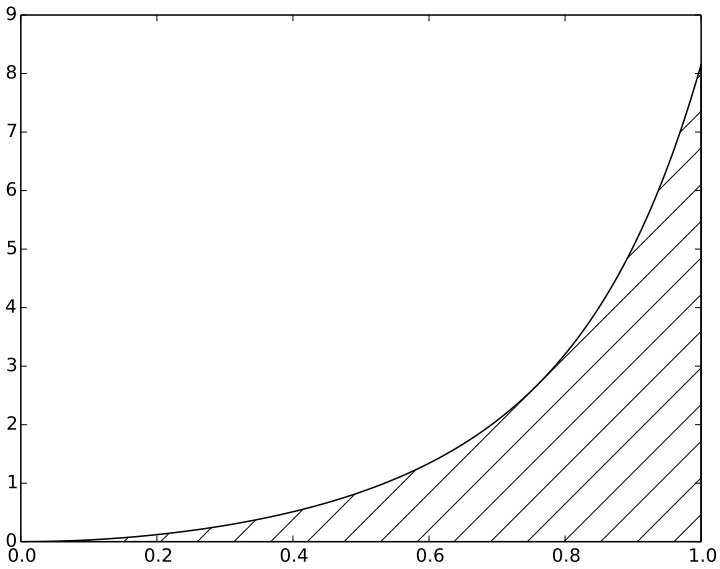


FIGURE 5.1 – L'intégrale de $v(t)$ interprétée comme l'aire sous le graphique de v.

naissance à 4 trapèzes, couvrant les intervalles de temps $[0, 0.2)$, $[0.2, 0.6)$, $[0.6, 0.8)$ et $[0.8, 1.0]$. Notez que nous en avons profité pour démontrer les calculs avec des intervalles de temps de tailles différentes.

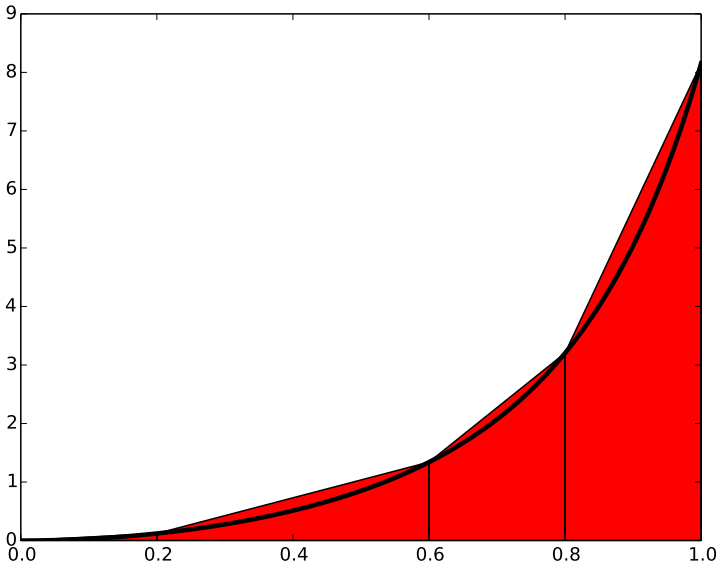


FIGURE 5.2 – Calculer approximativement l'intégrale d'une fonction comme la somme des aires des trapèzes.

Les aires des 4 trapèzes représentés sur la figure 5.2 constituent maintenant notre approximation

de l'intégrale (5.6) :

$$\int_0^1 v(t)dt \approx h_1(\frac{v(0)+v(0.2)}{2}) + h_2(\frac{v(0.2)+v(0.6)}{2}) + h_3(\frac{v(0.6)+v(0.8)}{2}) + h_4(\frac{v(0.8)+v(1.0)}{2}) \tag{5.8}$$

où

$$h_1 = (0.2-0.0) \tag{5.9}$$
$$h_2 = (0.6-0.2) \tag{5.10}$$
$$h_3 = (0.8-0.6) \tag{5.11}$$
$$h_4 = (1.0-0.8) \tag{5.12}$$

Avec $v(t) = 3t^2e^{t^3}$, chaque terme dans (5.8) est facilement calculé et notre calcul approximatif donne

$$\int_0^1 v(t)dt \approx 1.895 \tag{5.13}$$

Par rapport à la vraie réponse de 1.718, cela est d'environ 10%. Cependant, notez que nous avons utilisé seulement 4 trapèzes pour approximer la zone. Avec plus de trapèzes, l'approximation serait devenue meilleure, puisque les segments de droite du côté supérieur des trapèzes suivraient alors le graphique de plus près. Faire un autre calcul avec plus de trapèzes n'est pas trop tentant pour un humain paresseux, mais c'est un travail parfait pour un ordinateur ! Dérivons donc les expressions d'approximation de l'intégrale par un nombre arbitraire de trapèzes.

5.3.1 La formule générale

Pour une fonction donnée $f(x)$, nous voulons approximer l'intégrale $\int_a^b f(x)dx$ par n trapèzes (de largeur égale). Nous commençons par (5.4) et approchons chaque intégrale du côté droit avec un seul trapèze. En détail,

$$\begin{aligned}\int_a^b f(x)\,dx &= \int_{x_0}^{x_1} f(x)dx + \int_{x_1}^{x_2} f(x)dx + \ldots + \int_{x_{n-1}}^{x_n} f(x)dx, \\ &\approx h\frac{f(x_0)+f(x_1)}{2} + h\frac{f(x_1)+f(x_2)}{2} + \ldots + \\ &\quad h\frac{f(x_{n-1})+f(x_n)}{2}\end{aligned} \tag{5.14}$$

En simplifiant le côté droit de (5.14), nous obtenons

$$\int_a^b f(x)\,dx \approx \frac{h}{2}\left[f(x_0) + 2f(x_1) + 2f(x_2) + \ldots + 2f(x_{n-1}) + f(x_n)\right] \tag{5.15}$$

qui est écrit de façon plus compacte comme

$$\int_a^b f(x)\,dx \approx h\left[\frac{1}{2}f(x_0) + \sum_{i=1}^{n-1} f(x_i) + \frac{1}{2}f(x_n)\right] \tag{5.16}$$

Définition

Règles d'intégration composites : Le mot composite est souvent utilisé lorsqu'une méthode d'intégration numérique est appliquée avec plus d'un sous-intervalle. à vrai dire alors, écrire, par exemple, "la méthode du trapèze", devrait impliquer l'utilisation d'un seul trapèze, tandis que "la méthode du trapèze composite" est le nom le plus correct lorsque plusieurs trapèzes sont utilisés. Cependant, cette convention de dénomination n'est pas toujours suivie, donc dire que "la méthode du trapèze" peut pointer vers un seul trapèze ainsi que la règle composite avec de nombreux trapèzes.

5.3.2 Implémentation

Implémentation spécifique ou générale ? Supposons que notre objectif principal était de calculer l'intégrale spécifique $\int_0^1 v(t)dt$ avec $v(t) = 3t^2e^{t^3}$. D'abord, nous avons joué avec un simple calcul de main pour voir de quoi il s'agissait, avant de développer (comme c'est souvent le cas en mathématiques) une formule générale (5.16) pour l'intégrale générale ou «abstraite» $\int_a^b f(x)dx$. Pour résoudre notre problème spécifique $\int_0^1 v(t)dt$, nous devons ensuite appliquer la formule générale (5.16) aux données données (fonction et limites intégrales) dans notre problème. Bien que simples en principe, les étapes pratiques sont déroutantes pour beaucoup car la notation dans le problème abstrait de (5.16) diffère de la notation dans notre problème spécial. Clairement, les f, x et h dans (5.16) correspondent à v, t et peut-être Δt pour la largeur du trapèze dans notre problème spécial.

Note:

1. Faut-il écrire un programme spécial pour l'intégrale spéciale, en utilisant les idées de la règle générale (5.16), mais en remplaçant f par v, x par t et h par Δt ?
2. Faut-il implémenter la méthode générale (5.16) telle qu'elle se présente dans une fonction générale `trapeze(f, a, b, n)` et résoudre le problème spécifique en question par un appel spécialisé à cette fonction ?

L'alternative 2 est toujours le meilleur choix !

La première alternative dans l'encadré ci-dessus semble moins abstraite et donc plus attrayante pour beaucoup. Néanmoins, comme nous l'espérons, cela sera évident à partir des exemples, la deuxième alternative est en fait la plus simple et la plus fiable d'un point de vue mathématique et de programmation. Ces auteurs affirmeront que la deuxième alternative est l'essence même du pouvoir des mathématiques, tandis que la première alternative est la source de beaucoup de confusion sur les mathématiques !

Implémentation avec fonctions. Pour l'intégrale $\int_a^b f(x)dx$ calculée par la formule (5.16), nous voulons que le trapèze de la fonction Python correspondante prenne tout f, a, b et n en entrée et renvoie l'approximation à l'intégrale.

Nous écrivons une fonction Python `trapeze()` dans un fichier `trapeze_integral.py` aussi proche que possible de la formule (5.16), en nous assurant que les noms de variables correspondent à la notation mathématique :

```
## NOM DU PROGRAMME: trapeze_integral.py
def trapeze(f, a, b, n):
    h = (b-a)/n
    result = 0.5*f(a) + 0.5*f(b)
    for i in range(1, n):
        xi = a + i*h
        result += f(xi)
    result *= h
    return result
```

Résoudre notre problème spécifique en une session. Le simple fait d'avoir la fonction `trapeze()` comme seul contenu d'un fichier `trapeze_integral.py` fait automatiquement de ce fichier un module que nous pouvons importer et tester dans une session interactive :

```
In [3]: from trapeze_integral import trapeze
In [4]: from math import exp
In [5]: v = lambda t: 3*(t**2)*exp(t**3)
In [6]: n = 4
In [7]: numerical = trapeze(v, 0, 1, n)
In [8]: numerical
Out[8]: 1.9227167504675762
```

Calculons l'expression exacte et l'erreur dans l'approximation :

```
In [9]: V = lambda t: exp(t**3) - 1
In [10]: exact = V(1) - V(0)
In [11]: exact - numerical
Out[11]: -0.20443492200853108
```

Cette erreur est-elle convaincante ? On peut essayer un n plus grand :

```
In [12]: numerical = trapeze(v, 0, 1, n=400)
In [13]: exact - numerical
Out[13]: -2.1236490512777095e-05
```

Heureusement, beaucoup plus de trapèzes donnent une erreur beaucoup plus petite.

Résoudre notre problème spécifique dans un programme. Au lieu de calculer notre problème spécial dans une session interactive, nous pouvons le faire dans un programme. Comme toujours, un morceau de code faisant une chose particulière est mieux isolé en tant que fonction même si nous ne voyons aucune raison future d'appeler la fonction plusieurs fois et même si nous n'avons pas besoin d'arguments pour paramétrer ce qui se passe à l'intérieur de la fonction. Dans le cas présent, nous mettons simplement les instructions que nous aurions autrement mises dans un programme principal, à l'intérieur d'une fonction :

```
def application():
    from math import exp
    v = lambda t: 3*(t**2)*exp(t**3)
    n = int(input('n: '))
    numerical = trapeze(v, 0, 1, n)

    # Comparer avec le résultat exact
    V = lambda t: exp(t**3) - 1
    exact = V(1) - V(0)
    print(exact)
    error = exact - numerical
    print('n=%d: %.16f, erreur: %g' % (n, numerical, error))
```

Maintenant, nous calculons notre problème spécial en appelant `application()` comme la seule instruction du programme principal.

Faire un module. Lorsque nous avons les différentes parties de notre programme comme une collection de fonctions, il est très simple de créer un *module* qui peut être importé dans d'autres programmes. Ce fait, avoir notre code comme module, signifie que la fonction `trapeze()` peut facilement être réutilisée par d'autres programmes pour résoudre d'autres problèmes. Les exigences d'un module sont simples : mettez tout à l'intérieur des fonctions et laissez les appels de fonction dans le programme principal être dans le soi-disant *bloc de test* :

```
if __name__ == '__main__':
    application()
```

Le test `if` est vrai si le fichier de module, `trapeze_integral.py`, est exécuté en tant que programme et faux si le module est importé dans un autre programme. Par conséquent, lorsque nous effectuons une importation : `from trapeze_integral import trapeze` dans un fichier, le test échoue et `application()` n'est pas appelée, c'est-à-dire que notre problème spécial n'est pas résolu et n'imprime rien à l'écran. D'un autre côté, si nous exécutons `trapeze_integral.py` dans la fenêtre du terminal, la condition de test est positive, `application()` est appelée et nous obtenons une sortie dans la fenêtre :

```
Terminal> python trapeze_integral.py
n: 400
n=400: 1.7183030649495579, error: -2.12365e-05
```

5.4 La méthode du point milieu composite

5.4.1 L'idée

Plutôt que d'approximer l'aire sous une courbe par des trapèzes, nous pouvons utiliser des rectangles simples. Il peut sembler moins précis d'utiliser des lignes horizontales et non des lignes obliques suivant la fonction à intégrer, mais une méthode d'intégration basée sur des rectangles (la méthode du point milieu) est en fait légèrement plus précise que celle basée sur des trapèzes !

Dans la méthode du milieu, nous construisons un rectangle pour chaque sous-intervalle où la hauteur est égale à f au milieu du sous-intervalle. Faisons-le pour quatre rectangles, en utilisant les mêmes sous-intervalles que nous avions pour les calculs manuels avec la méthode du trapèze : $[0, 0.2)$, $[0.2, 0.6)$, $[0.6, 0.8)$ et $[0.8, 1.0]$. On a

$$\int_0^1 f(t)dt \approx h_1 f\left(\frac{0+0.2}{2}\right) + h_2 f\left(\frac{0.2+0.6}{2}\right) + h_3 f\left(\frac{0.6+0.8}{2}\right) + h_4 f\left(\frac{0.8+1.0}{2}\right) \tag{5.17}$$

où h_1, h_2, h_3 et h_4 sont les largeurs des sous-intervalles, utilisées précédemment avec la méthode du trapèze et définies dans (5.9)-(5.12).

Avec $f(t) = 3t^2 e^{t^3}$, l'approximation devient 1.632. Comparé à la vraie réponse (1.718), c'est environ 5% trop petit, mais c'est mieux que ce que nous avons obtenu avec la méthode trapézoïdale (10%) avec les mêmes sous-intervalles. Plus de rectangles donnent une meilleure approximation.

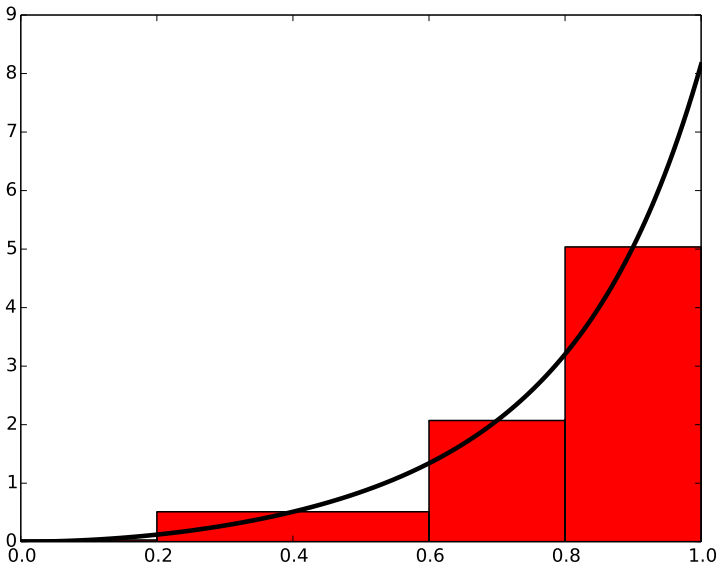


FIGURE 5.3 – Calcul approximatif de l'intégrale d'une fonction comme la somme des aires des rectangles.

5.4.2 La formule générale

Dérivons une formule pour la méthode du milieu basée sur n rectangles d'égale largeur :

$$\begin{aligned}\int_a^b f(x)\,dx &= \int_{x_0}^{x_1} f(x)dx + \int_{x_1}^{x_2} f(x)dx + \ldots + \int_{x_{n-1}}^{x_n} f(x)dx, \\ &\approx hf\left(\frac{x_0+x_1}{2}\right) + hf\left(\frac{x_1+x_2}{2}\right) + \ldots + hf\left(\frac{x_{n-1}+x_n}{2}\right) \qquad (5.18) \\ &\approx h\left(f\left(\frac{x_0+x_1}{2}\right) + f\left(\frac{x_1+x_2}{2}\right) + \ldots + f\left(\frac{x_{n-1}+x_n}{2}\right)\right) \qquad (5.19)\end{aligned}$$

Cette somme peut être écrite de façon plus compacte comme

$$\int_a^b f(x)dx \approx h\sum_{i=0}^{n-1} f(x_i) \qquad (5.20)$$

où $x_i = \left(a+\frac{h}{2}\right) + ih$.

5.4.3 Implémentation

Nous suivons les conseils et les enseignements tirés de l'implémentation de la méthode trapèze et réalisons une fonction `midpoint(f, a, b, n)` (dans un fichier `midpoint_integral.py`) pour implémenter la formule générale (5.20) :

```
## NOM DU PROGRAMME: midpoint_integral.py
def midpoint(f, a, b, n):
```

```
    h = float(b-a)/n
    result = 0
    for i in range(n):
        xi = (a + h/2.0) + i*h
        result += f(xi)
    result *= h
    return result
```

Nous pouvons tester la fonction comme nous l'avons expliqué pour la méthode du trapèze similaire. L'erreur dans notre problème particulier $\int_0^1 3t^2 e^{t^3} dt$ avec quatre intervalles est maintenant d'environ 0.1 contrairement à 0.2 pour la règle du trapèze. Les différences sont rarement d'une importance pratique, et sur un ordinateur portable, nous pouvons facilement utiliser $n = 10^6$ et obtenir la réponse avec une erreur d'environ 10^{-12} en quelques secondes.

5.4.4 Comparaison des méthodes du trapèze et du point milieu

L'exemple suivant montre la facilité avec laquelle nous pouvons combiner les fonctions `trapeze()` et `midpoint()` pour comparer les deux méthodes dans le fichier `compare_integration_methods.`

```
## NOM DU PROGRAMME: compare_integration_methods.py
#% IMPORTATION
from trapeze_integral import trapeze
from midpoint_integral import midpoint
from math import exp

g = lambda y: exp(-y**2)
a = 0
b = 2
print("      n      point milieu     trapèze")
for i in range(1, 21):
    n = 2**i
    m = midpoint(g, a, b, n)
    t = trapeze(g, a, b, n)
    print('%7d %.16f %.16f'%(n, m, t))
```

Notez les efforts mis en forme agréable - la sortie devient

```
      n      point milieu     trapèze
      2 0.8842000076332692 0.8770372606158094
      4 0.8827889485397279 0.8806186341245393
      8 0.8822686991994210 0.8817037913321336
     16 0.8821288703366458 0.8819862452657772
     32 0.8820933014203766 0.8820575578012112
     64 0.8820843709743319 0.8820754296107942
    128 0.8820821359746071 0.8820799002925637
    256 0.8820815770754198 0.8820810181335849
    512 0.8820814373412922 0.8820812976045025
   1024 0.8820814024071774 0.8820813674728968
   2048 0.8820813936736116 0.8820813849400392
   4096 0.8820813914902204 0.8820813893068272
   8192 0.8820813909443684 0.8820813903985197
  16384 0.8820813908079066 0.8820813906714446
  32768 0.8820813907737911 0.8820813907396778
 131072 0.8820813907631487 0.8820813907610036
 262144 0.8820813907625702 0.8820813907620528
 524288 0.8820813907624605 0.8820813907623183
1048576 0.8820813907624268 0.8820813907623890
```

Une inspection visuelle des chiffres montre à quelle vitesse les chiffres se stabilisent dans les deux méthodes. Il semble que 13 chiffres se soient stabilisés dans les deux dernières lignes.

Note: Les méthodes du trapèze et du point milieu ne sont que deux exemples dans une jungle de règles d'intégration numérique. D'autres méthodes célèbres sont la règle de Simpson et la quadrature de Gauss. Ils fonctionnent tous de la même manière :

$$\int_a^b f(x)dx \approx \sum_{i=0}^{n-1} w_i f(x_i)$$

Autrement dit, l'intégrale est approximée par une somme d'évaluations de fonctions, où chaque évaluation $f(x_i)$ reçoit un poids w_i. Les différentes méthodes diffèrent par la façon dont elles construisent les points d'évaluation x_i et les poids w_i. Nous avons utilisé des points x_i également espacés, mais une précision plus élevée peut être obtenue en optimisant l'emplacement de x_i.

5.5 Intégration Monte Carlo

Les méthodes de Monte Carlo sont des techniques de calcul probabilistes. Au cœur, un algorithme de Monte Carlo représente de façon aléatoire certaines valeurs de l'espace de valeurs d'un paramètre considéré. La combinaison de plusieurs paramètres permet de tirer des conclusions stochastiques des relations. L'intégration des fonctions mathématiques de la forme :

$$A = \int_a^b f(x) \cdot dx$$

Pour effectuer une intégration, nous voulons savoir comment les valeurs sélectionnées au hasard sont réparties : lesquelles des valeurs sont égales ou inférieures à la valeur de la fonction et lesquelles sont supérieures. Il s'agit d'une décision binaire qui divise les valeurs aléatoires en deux groupes. Du rapport de la taille des groupes, nous pouvons tirer nos conclusions.

Nous utilisons la fonction (intégrande) comme critère de décision uniquement. L'algorithme ne nous fournit rien d'autre que des comptes/fréquences. La fermeture probabiliste est alors :

$$\frac{\text{cas favorables}}{\text{cas possibles}} = \frac{n}{N} = \frac{A_{sous\,la\,fonction}}{A_{aire\,totale}}$$

La zone $A_{sous\,la\,fonction}$ est la zone inconnue qui nous intéresse. Pour $A_{aire\,totale}$, nous choisissons arbitrairement une région simple, cette zone que nous pouvons calculer sans difficultés.

5.5.1 Exemple : détermination de π

À titre d'exemple, nous choisissons un cercle dont la fonction mathématique est donnée par la première :

$$x^2 + y^2 = R^2$$
$$y = f(x) = \sqrt{R^2 - x^2}$$

Pour l'estimation de π, l'aire du cercle est comparée à l'aire du carré $2R \times 2R$, ce rapport est $\pi/4$:

$$A_{cercle} = R^2 \cdot \pi$$
$$A_{carré} = (2R)^2 = 4R^2$$

Nous générons au hasard (x_rand, y_rand)-points. Pour chaque point, nous devons décider s'il se trouve à l'intérieur ou à l'extérieur du cercle. Pour cela, nous utilisons la différence $y_{rand} - f(x_{rand})$, où $f(x) = \sqrt{R^2 - x^2}$ est la fonction d'un cercle dans le premier quadrant. Nous pouvons compter le nombre de points à l'intérieur du cercle. Nous pouvons compter le nombre de points à l'intérieur du cercle. Le rapport n/N est supposé être égal au rapport $A_{cercle}/A_{carré}$

$$\frac{\text{n = (x,y)-points dans le cercle}}{\text{N = (x,y)-points dans le carré}} = \frac{A_{cercle}}{A_{carré}} = \frac{R^2 \cdot \pi}{4R^2} = \frac{\pi}{4}$$
$$\pi = 4\frac{n}{N}$$

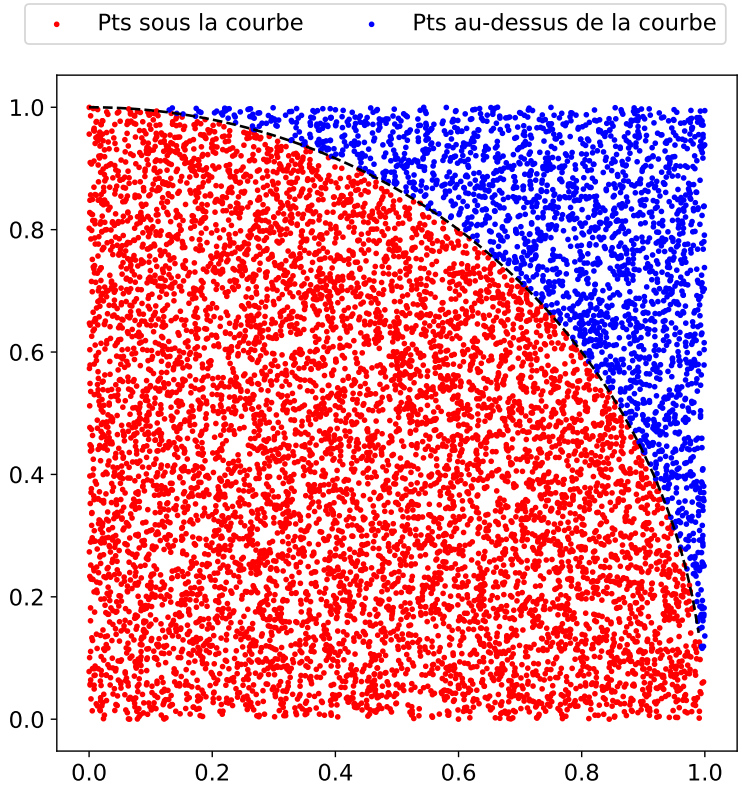
5.5.2 implémentation

Nous avons vu l'intégration de Monte Carlo lorsque nous avons calculé $\pi/4$ en calculant l'aire du quart de cercle unitaire.

Voici le code :

```
## NOM DU PROGRAMME: MC_integral.py
#% IMPORTATION
import numpy as np
import matplotlib.pyplot as plt

def f(x):
    '''
    fonction pour un cercle
    '''
    return np.sqrt(1-x*x)

N = 10000   # nombre d'essais
x0 = 0
x1 = 1

x = np.arange(x0, x1, 0.01)
y = f(x)
fmax = max(y)
np.random.seed(6)
x_rand = x0 + (x1 - x0) * np.random.rand(N)
y_rand = fmax * np.random.rand(N)
n = np.sum(y_rand - f(x_rand) < 0.0) # nombre de points dans le cercle
#----- Sortie et graphiques -------------------
print('PI numpy       : ', np.pi)
print('PI monte carlo : ', 4*n/N)
print('différence     : ', 4*n/(N) - np.pi)

index_below = np.where(y_rand < f(x_rand))
index_above = np.where(y_rand >= f(x_rand))
plt.figure(figsize=(7,7))
plt.plot(x,f(x),'--k')
plt.scatter(x_rand[index_below], y_rand[index_below],
            c="r", s = 5, label = "Pts sous la courbe")
plt.scatter(x_rand[index_above], y_rand[index_above],
            c="b", s = 5, label = "Pts au-dessus de la courbe")
plt.legend(bbox_to_anchor=(0., 1.02, 1., .102), ncol=2)
plt.show()
```

```
PI numpy       :  3.141592653589793
PI monte carlo :  3.1436
différence     :  0.002007346410207056
```



Nous pouvons généraliser cette approche aux courbes autres que $y = \sqrt{1 - x^2}$. L'idée est la suivante : pour une courbe arbitraire, trouvez le rectangle qui la contient, générez un point aléatoire dans ce rectangle et déterminez combien de points aléatoires se trouvent sous la courbe.

5.6 Travaux dirigés

5.6.1 Exercice 1 : Vitesse d'une fusée

On lance une fusée verticalement du sol et l'on mesure pendant les premières 80 secondes l'accélération γ :

t[s]	0	10	20	30	40	50	60	70	80
γ [m s^{-2}]	30	31.63	33.44	35.47	37.75	40.33	43.29	46.70	50.67

Calculer la vitesse V de la fusée à l'instant $t = 80$ s, par la méthode des trapèzes.

5.6.2 Exercice 2 : Valeur approchée de π

Étant donnée l'égalité :

$$\pi = 4\left(\int_0^\infty e^{-x^2}dx\right)^2 = 4\left(\int_0^{10} e^{-x^2}dx + \epsilon\right)^2 \tag{5.21}$$

avec $0 < \epsilon < 10^{-44}$, utiliser la méthode des trapèzes composite à 10 intervalles pour estimer la valeur de π.

5.6.3 Exercice 3 : Intégration adaptative

Supposons que nous voulons utiliser la méthode des trapèzes ou du point milieu pour calculer une intégrale $\int_a^b f(x)dx$ avec une erreur inférieure à une tolérance prescrite ϵ. Quelle est la taille appropriée de n ?

Pour répondre à cette question, nous pouvons entrer une procédure itérative où nous comparons les résultats produits par n et $2n$ intervalles, et si la différence est inférieure à ϵ, la valeur correspondant à $2n$ est retournée. Sinon, nous avons n et répétons la procédure.

Indication. Il peut être une bonne idée d'organiser votre code afin que la fonction `integration_adaptive` peut être utilisé facilement dans les programmes futurs que vous écrivez.

a) Écrire une fonction `integration_adaptive(f, a, b, eps, method="midpoint")` qui implémente l'idée ci-dessus (`eps` correspond à la tolérance ϵ, et la méthode peut être `midpoint` ou `trapeze`).

b) Testez la méthode sur $g(x) = \int_0^2 \sqrt{x}dx$ pour $\epsilon = 10^{-1}, 10^{-10}$ et notez l'erreur exacte.

c) Faites un tracé de n en fonction de $\epsilon \in [10^{-1}, 10^{-10}]$ pour $\int_0^2 \sqrt{x}dx$. Utilisez l'échelle logarithmique pour ϵ.

Remarques. Le type de méthode exploré dans cet exercice est appelé *adaptatif*, car il essaie d'adapter la valeur de n pour répondre à un critère d'erreur donné. La vraie erreur peut très rarement être calculée (car nous ne connaissons pas la réponse exacte au problème de calcul), il faut donc trouver d'autres indicateurs de l'erreur, comme celui ici où les changements de la valeur intégrale, comme le nombre d'intervalles est doublé, est pris pour refléter l'erreur.

5.6.4 Exercice 4 : Intégration de x élevé à x

Considérons l'intégrale

$$I = \int_0^2 x^x \, dx \, .$$

L'intégrande x^x n'a pas de primitive qui peut être exprimé en termes de fonctions standard (visitez `http://wolframalpha.com` et tapez `integral x^x dx from 0 to 2` pour vous

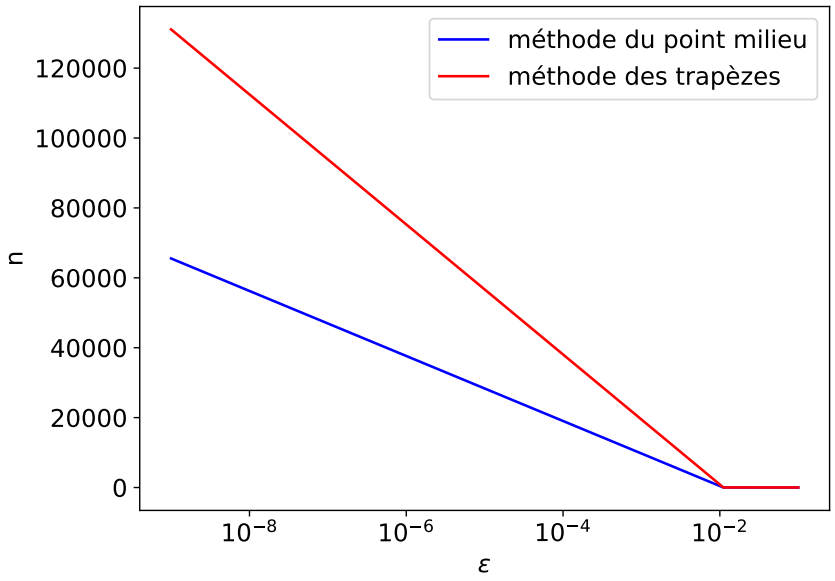


FIGURE 5.4 – Affichage de n en fonction de ϵ lorsque $\int_0^2 \sqrt{x}dx$ est calculé par la méthode du point milieu (bleu) et la méthode des trapèzes (rouge).

convaincre que notre affirmation est juste. Notez que Wolfram alpha vous donne une réponse, mais cette réponse est une approximation, elle n'est pas *exacte*. C'est parce que Wolfram alpha utilise également des méthodes numériques pour arriver à la réponse, comme vous le ferez dans cet exercice). Par conséquent, nous sommes obligés de calculer l'intégrale par des méthodes numériques. Calculez un résultat composé de quatre chiffres.

Indication. Utilisez des idées de l'exercice 5.6.3.

5.6.5 Exercice 5 : Orbitales atomiques

Pour décrire la trajectoire d'un électron autour d'un noyau, une description probabiliste est adoptée : l'électron n'est plus caractérisé par ses coordonnées spatiales mais par sa *probabilité de présence* en un point de l'espace.

Pour simplifier le problème, on considérera que cette probabilité de présence ne dépend que de la variable r, distance entre l'électron et le centre du noyau. Pour une orbitale $1s$, la probabilité de trouver l'électron entre les rayons r_1 et r_2 s'écrit :

$$P_{s1} = \int_{r_1}^{r_2} \underbrace{4 \times \frac{r^2}{a_0^3} \times e^{-2\times\frac{r}{a_0}}}_{\text{densité radiale}} dr$$

avec $a_0 = 0.529$ Å, appelé le rayon de Bohr.

La densité radiale, représentée dans la figure 5.5, est maximale pour $r = a_0$. Ce rayon qui maximise la densité radiale est appelé le *rayon orbitalaire*.

Note: Dans ce problème, les distances seront conservées en Angström.

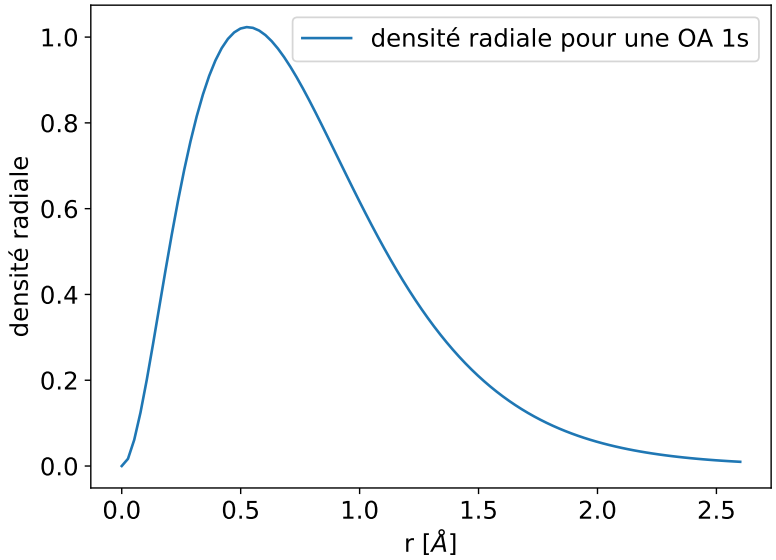


FIGURE 5.5 – Densité radiale pour une orbitale atomique 1s.

a) Définir une fonction `densite_radiale()`, définie entre 0 et ∞ qui prend comme paramètre variable un rayon r et comme paramètre par défaut $a_0 = 0.529$ Å et renvoie la valeur $4 \times \frac{r^2}{a_0^3} \times e^{-2\times\frac{r}{a_0}}$.

b) Tracer la densité radiale pour $r \in [0, 2.6]$ Å, afin d'obtenir le même graphique sur la figure 5.5.

c) On souhaite déterminer la probabilité de présence de l'électron entre 0 et a_0. Évaluer cette probabilité à l'aide de 100 rectangles. On pourra vérifier que la réponse obtenue est proche de 0.32.

d) Déterminer le nombre entier n, tel que l'électron ait une probabilité supérieure ou égale à 90% de se trouver entre 0 et $n * a_0$.

e) On souhaite désormais évaluer la probabilité de trouver l'électron proche du rayon de Bohr, c'est-à-dire entre $0.9 * a_0$ et $1.1 * a_0$. Évaluer cette probabilité à l'aide de 100 rectangles.

f) D'après la valeur obtenue à la question précédente, que penser de la description des trajectoires des électrons par orbite autour du noyau ?

Note: On répondra en commentaire dans le programme.

Chapitre 6

Équations différentielles ordinaires

6.1 Introduction

Dans les domaines scientifiques et industriels, il est courant aujourd'hui d'étudier la nature ou les dispositifs technologiques au moyen de modèles sur ordinateur. Avec de tels modèles, l'ordinateur agit comme un laboratoire virtuel où les expériences peuvent être effectuées de manière rapide, fiable, sûre et économique.

Les équations différentielles constituent l'un des outils mathématiques les plus puissants pour comprendre et prédire le comportement des systèmes dynamiques de la nature, de l'ingénierie et de la société. Un système dynamique est un système avec un état, généralement exprimé par un ensemble de variables, évoluant dans le temps. Par exemple, un pendule oscillant, la propagation d'une maladie et les conditions météorologiques sont des exemples de systèmes dynamiques. Nous pouvons utiliser les lois fondamentales de la physique, ou l'intuition simple, pour exprimer des règles mathématiques qui régissent l'évolution du système dans le temps. Ces règles prennent la forme d'équations différentielles.

6.2 Exemple I : Radioactivité

6.2.1 La découverte de la radioactivité

La radioactivité a été découverte en France, de 1896 à 1898, par Henri Becquerel, qui a mis en évidence l'existence d'un rayonnement invisible provenant de l'uranium (voir Figure 6.1), et par Pierre et Marie Curie qui ont montré la généralité de ce phénomène, lui ont donné son nom, et découvert deux éléments chimiques particulièrement radioactifs, le polonium et le radium. Dans l'histoire de cette découverte, et du développement de toutes ses conséquences, on retrouve toutes les grandes questions liées à la recherche, aux mécanismes de la découverte, aux remises en cause des acquis de la science et à l'exploitation scientifique, technologique et industrielle des connaissances nouvelles.

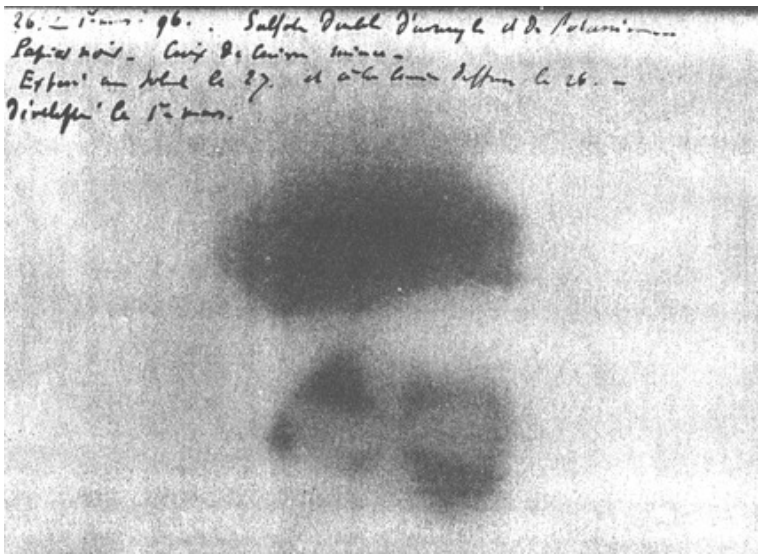


FIGURE 6.1 – Cliché développé par Becquerel le 1^{er} mars 1896 après être resté dans un tiroir. On distingue dans la tache inférieure, une croix de Malte à laquelle Becquerel fait allusion (« Si, entre la lamelle du sel d'uranium et la lame d'aluminium ou le papier noir, on interpose un écran formé d'une lame de cuivre ... par exemple en forme de croix, on observe dans l'image la silhouette de cette croix, en plus clair... »). Les annotations sont de la main de Becquerel.[s :OpenEdition Journals, Henri Becquerel : découverte de la radioactivité]

6.2.2 Loi de désintégration radioactive

Considérons la désintégration radioactive des noyaux. Le nombre de noyaux, N, suit l'équation différentielle ordinaire :

$$\frac{dN(t)}{dt} = -\frac{N(t)}{\tau} \tag{6.1}$$

où τ est la constante de temps de décroissance (on l'appelle aussi durée de vie moyenne). Cette équation peut être intégrée directement, avec la solution :

$$N(t) = N_0 e^{-t/\tau} \tag{6.2}$$

mais nous voulons essayer de résoudre l'équation numériquement.

L'approche la plus simple consiste à exprimer le nombre de noyaux à l'instant $t+\Delta t$ en termes de nombre à l'instant t :

$$N(t+\Delta t) = N(t) - \frac{N(t)}{\tau}\Delta t + \mathcal{O}(\Delta t^2) \tag{6.3}$$

Si nous commençons par N_0 noyaux à l'instant $t=0$, alors à $t=\Delta t$ nous aurons $N(\Delta t) \approx N_0 - (N_0/\tau)\Delta t$; at $t = 2\Delta t$ nous aurons $N(2\Delta t) \approx N(\Delta t) - [N(\Delta t)/\tau]\Delta t$ etc. L'erreur de troncature est $\mathcal{O}(\Delta t^2)$. Par conséquent, si la taille du pas Δt est petite, nous nous attendons à ce que notre solution numérique soit proche de la solution exacte. Cette méthode d'intégration d'une équation différentielle ordinaire est connue sous le nom de **méthode d'Euler**.

Voici un programme qui implémentera cette méthode d'intégration de l'équation différentielle pour la désintégration radioactive :

```python
## NOM DU PROGRAMME: desintegration.py
#%% IMPORTATION
import numpy as np
import matplotlib.pyplot as plt
#%% SOLUTION EXACTE
def n_exact(t, noyaux0, tau):
    return noyaux0*np.exp(-t/tau)
#%% ENTRÉES
noyaux0 = int(input("nombre initial de noyaux: "))
tau = float(input('constante de temps de décroissance: '))
dt = float(input('pas de temps: '))
tmax = int(input('temps de fin de la simulation: '))
nsteps = int(tmax/dt)
noyaux = np.zeros(nsteps)
t = np.zeros(nsteps)
#%% VLEURS INITIALES
t[0] = 0.0
noyaux[0] =noyaux0
#%% BOUCLE PRINCIPALE: MÉTHODE D'EULER
for i in range(nsteps-1):
    t[i+1] = t[i] + dt
    noyaux[i+1] = noyaux[i] - noyaux[i]/tau*dt
#%% TRAÇAGE DU GRAPHIQUE
plt.figure(figsize=(8,5))
plt.plot(t, noyaux, '-r', label='Solution: Euler')
plt.plot(t, n_exact(t, noyaux0, tau), '--b', label='Solution exacte')
plt.xlabel('temps')
plt.ylabel('N(t)')
plt.title('Désintégration radioactive')
plt.grid()
plt.legend()
plt.tight_layout()
plt.savefig("desintegration.png")
plt.savefig("desintegration.pdf")
plt.show()
```

Le programme demande le nombre initial de noyaux, N_0, la constante de temps de décroissance τ, le pas de temps Δt et la durée totale de l'intégration t_{max}. Lorsque ce programme est exécuté avec les valeurs d'entrée sont ; $N_0 = 100$, $\tau = 1$, $\Delta t = 0.04$ et $t_{max} = 5$, le programme produit le tracé présenté dans la Figure 6.2.

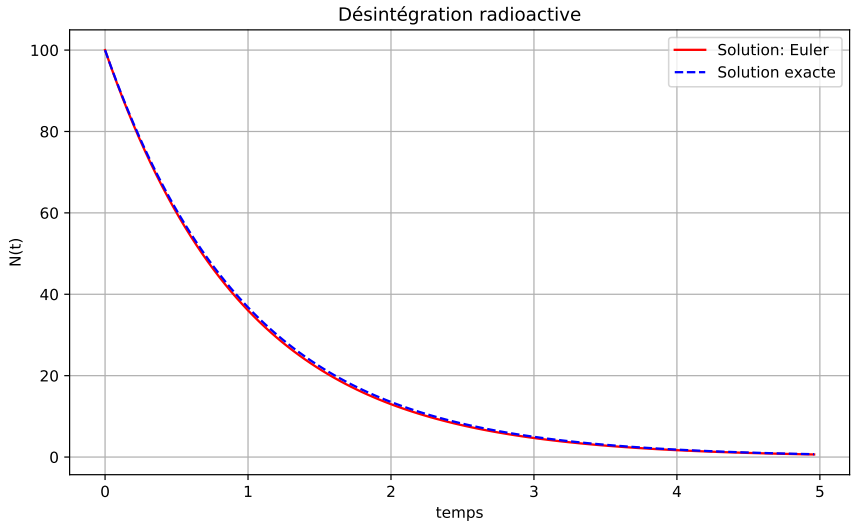


FIGURE 6.2 – Résultat de l'exécution du programme *desintegration.py* avec entrée $N_0 = 100$, $\tau = 1$, $\Delta t = 0.04$ et $t_{max} = 5$.

Voyons maintenant à quel point notre programme est proche de la solution exacte. Vraisemblablement, lorsque le pas Δt est grand, l'erreur sera pire ; aussi, les erreurs grandissent avec le temps. Pour voir cela, considérons une version modifiée de notre programme *desintegration.py* qui trace la différence fractionnaire entre le résultat numérique et le résultat exact donné par Eq. (6.2). Notre nouveau programme effectuera des évolutions numériques sur un certain nombre de **différentes valeurs du pas** afin que nous puissions voir comment l'erreur dépend du degré de raffinement de Δt.

```
## NOM DU PROGRAMME: desintegrationErr.py
#IMPORTATION
import numpy as np
import matplotlib.pyplot as plt
# ENTRÉES
noyaux0 = int(input("nombre initial de noyaux: "))
tau = float(input('constante de temps de décroissance: '))
dtbas = float(input('pas de temps de résolution le plus bas: '))
nres = int(input('nombre de raffinements de résolution: '))
tmax = int(input('temps de fin de la simulation: '))
# BOUCLE PRINCIPALE: CALCUL D'ERREURS
for n in range(nres):
    raffine = 10**n
    dt = dtbas/raffine
    nsteps = int(tmax/dt)
    noyaux = noyaux0
    err = np.zeros(nsteps)
    t = np.zeros(nsteps)
    # BOUCLE SECONDAIRE: MÉTHODE D'EULER
    for i in range(nsteps-1):
        t[i+1] = t[i] + dt
        noyaux = noyaux - noyaux/tau*dt
        exact = noyaux0 * np.exp(- t[i+1]/tau)
        err[i+1] = abs((noyaux - exact)/exact)
    # tracer l'erreur à cette résolution
    plt.loglog(t[raffine::raffine], err[raffine::raffine],
               '.-', label='dt = '+str(dt))

plt.legend(loc=4)
plt.xlabel('temps')
plt.ylabel('erreur fractionnaire')
plt.title("Erreur d'intégration de la désintégration radioactive")
plt.grid(linestyle='-', which='major')
plt.grid(which='minor')
plt.savefig("desintegrationErr.png")
plt.savefig("desintegrationErr.pdf")
plt.show()
```

Ce programme produit les résultats montrés à la Figure 6.3.

Les erreurs se rapprochent de manière linéaire avec le temps (les lignes du tracé logarithmique ont une pente approximativement égale à l'unité) et chaque facteur de 10 dans le raffinement diminue l'erreur fractionnaire d'un facteur 10. Pour comprendre cela, notez que le terme que nous avons jeté dans l'expansion de Taylor de notre équation différentielle ordinaire était le terme d^2N/dt^2, donc chaque étape introduit une erreur de :

$$e_i \approx \frac{1}{2}\frac{d^2N(t_i)}{dt^2}\Delta t^2 = \frac{N(t_i)}{2\tau^2}\Delta t^2 \tag{6.4}$$

Ceci est connu sous le nom **d'erreur locale**. Si l'erreur locale d'un schéma d'intégration numérique est $\mathcal{O}(\Delta t^{p+1})$ comme $t \to 0$, alors on dit que c'est l'ordre p. La méthode d'Euler est donc un schéma d'intégration de premier ordre. **L'erreur globale** est l'erreur accumulée

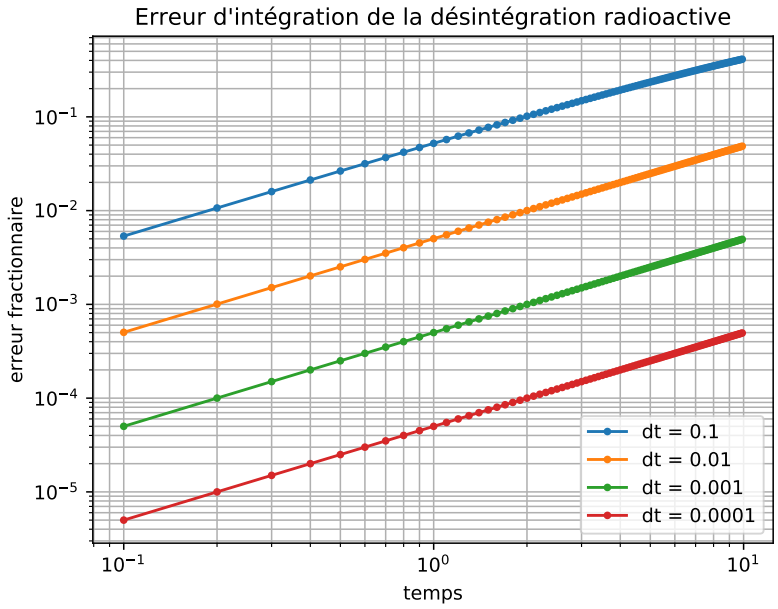


FIGURE 6.3 – Résultat de l'exécution du programme *desintegrationErr.py* avec entrée $N_0 = 100$, $\tau = 1$, $\Delta t = 0.1$, $N_{res} = 4$ et $t_{max} = 10$.

lorsque l'intégration est effectuée pendant une certaine durée T. Le nombre d'étapes requis est $n = T/\Delta t$ et chaque étape $i = 1...n$ accumule une erreur e_i, nous nous attendons donc à ce que l'erreur globale soit :

$$E_n \leq \sum_{i=1}^{n} e_i \leq T \frac{N_0}{2\tau^2} \Delta t \tag{6.5}$$

puisque $e_i \leq \frac{N_0}{2\tau^2} \Delta t^2$. Notez que pour un schéma d'intégration d'ordre p, l'erreur sera $\mathcal{O}(\Delta t^p)$; de plus, l'erreur grandit avec le temps T. Pour la méthode d'Euler, l'erreur croît de manière approximativement linéaire avec T et avec Δt, ce qui est ce que nous voyons sur la Figure 6.3.

> **Note:** La méthode d'Euler n'est pas une méthode recommandée pour résoudre des équations différentielles ordinaires. S'agissant simplement du premier ordre, une précision souhaitée n'est obtenue que pour de très petites valeurs de Δt, de nombreuses étapes d'intégration sont donc nécessaires pour faire évoluer le système pour une durée donnée T. Mais le coût en calcul de la méthode d'Euler n'est pas son seul inconvénient : elle n'est pas particulièrement stable non plus, comme nous le verrons plus loin dans ce chapitre..

6.3 Mouvement d'un projectile

Un autre exemple d'équation différentielle ordinaire est celui du mouvement du projectile, pour lequel les équations du mouvement sont :

$$\frac{dx}{dt} = v_x, \qquad \frac{dv_x}{dt} = 0, \tag{6.6}$$

$$\frac{dy}{dt} = v_y, \qquad \frac{dv_y}{dt} = -g, \tag{6.7}$$

où g est l'accélération de pesanteur. Nous pouvons utiliser la méthode d'Euler pour écrire chaque dérivée sous une forme de différence finie convenant à l'intégration numérique :

$$x_{i+1} = x_i + v_{x,i}\Delta t, \qquad v_{x,i+1} = v_{x,i}, \tag{6.8}$$

$$y_{i+1} = y_i + v_{y,i}\Delta t, \qquad v_{y,i+1} = v_{y,i} - g\Delta t, \tag{6.9}$$

Les trajectoires d'un projectile lancé avec une vitesse $v_0 = 10\ \ m\ s^{-1}$ à différents angles sont tracées par le programme `projectile.py` et sont tracées à la figure ci-dessous.

```
## NOM DU PROGRAMME: projectile.py
#%% IMPORTATION
import numpy as np
import matplotlib.pyplot as plt
#%% CONSTANTES
# accélération de la pesanteur(m/s^2)
g = 9.8
# angles de tire (deg)
angles = [30, 35, 40, 45, 50, 55]
# vitesse initiale (m/s())
v0 = 20.0
N = 10000 # Nombre de pas de temps
# pas de temps (s)
dt =0.001
#%% VLEURS INITIALES
x = np.zeros(N)
y = np.zeros(N)
vx = np.zeros(N)
vy = np.zeros(N)
for theta in angles:
    vx[0] = v0 * np.cos(theta*np.pi/180.0)
    vy[0] = v0 * np.sin(theta*np.pi/180.0)
    x[0], y[0]=0, 1
    # MÉTHODE D'EULER
    for i in range(N-1):
        x[i+1] = x[i] + vx[i] * dt
        y[i+1] = y[i] + vy[i] * dt
        vx[i+1] = vx[i]
        vy[i+1] = vy[i] - g * dt
    plt.plot(x, y, lw =2, label=str(theta)+' deg')
#%% GRAPHIQUE: TRAJECTOIRES
plt.xlabel('x [m]')
plt.ylabel('y [m]')
plt.suptitle("Trajectoire d'un projectile", weight='bold')
plt.title(r'Pas de temps: $\Delta_t$ = {:.3f} s'.format(dt))
plt.grid()
plt.legend()
plt.axis([0, 45, 0, 15])
# ENREGISTRER ET AFFICHER LA FIGURE
plt.savefig("projectile.png")
plt.savefig("projectile.pdf")
plt.show()
```

Nous voyons, comme prévu, que la plus grande plage est atteinte pour un angle de lancement de 45°.

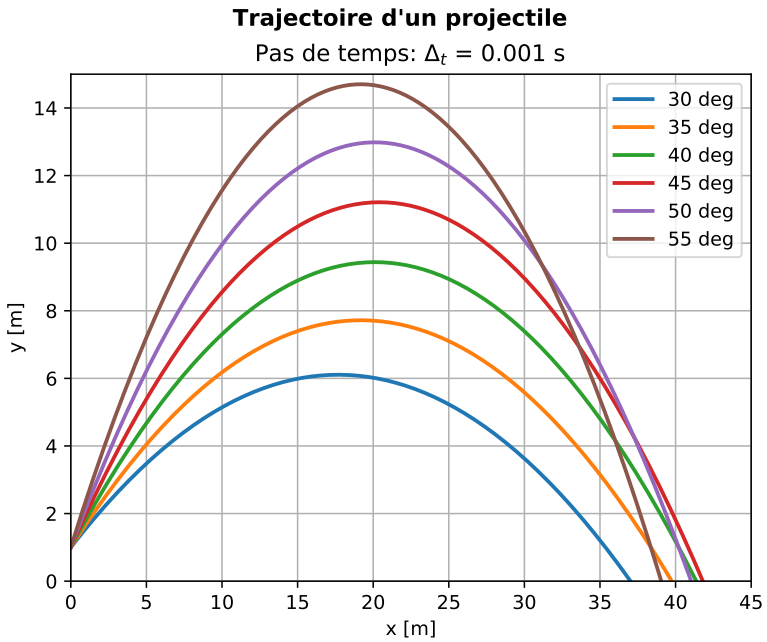


FIGURE 6.4 – Résultats de l'exécution du programme `projectile.py`. On voit que la plus grande plage est atteinte avec un angle de lancement de $\theta = 45°$.

Trouver la trajectoire d'un projectile compte tenu de ses conditions initiales, $v_{x,i}$ et $v_{x,i}$ ou de manière équivalente v_0 et θ, est relativement simple. Cependant, supposons que nous voulons trouver l'angle de lancement θ requis pour atteindre une cible à une distance donnée avec une vitesse initiale v_0 donnée. Ceci est un exemple de problème de la *valeur aux limites* à deux points.Une approche pour résoudre un tel problème est connue comme *méthode de tir*.

L'idée est simple : devinez la valeur de θ, effectuez l'intégration, déterminez combien vous manquez votre note, puis affinez votre estimation de manière itérative jusqu'à ce que vous soyez suffisamment proche de la cible. Si $\Delta x(\theta)$ est la quantité que vous manquez la cible avec l'angle de lancement θ alors l'objectif est de résoudre l'équation :

$$\Delta x(\theta) = 0 \tag{6.10}$$

pour θ. Ce problème général s'appelle **la recherche de racine**. Nous allons utiliser ici une méthode assez simple pour résoudre une racine appelée *méthode de bissection*. Supposons que nous savons que la racine de l'équation (6.10) se situe quelque part dans l'intervalle $\theta_1 < \theta < \theta_2$ et $\Delta x(\theta_1)$ a le signe opposé de $\Delta x(\theta_2)$ (c'est-à-dire si $\Delta x(\theta_1) < 0$ alors $\Delta x(\theta_2) > 0$, ou vice versa). On dit alors que θ_1 et θ_2 *encadrent* la racine.

Commençons par évaluer $\Delta x(\theta_1)$, $\Delta x(\theta_2)$ et $\Delta x(\theta_{deviner})$ avec $\theta_{deviner}$ au milieu entre θ_1 et θ_2 , $\theta_{deviner} = \frac{1}{2}(\theta_1 + \theta_2)$. Si le signe de $\Delta x(\theta_{deviner})$ est identique au signe de $\Delta x(\theta_1)$, alors nous savons que la racine doit être comprise entre $\theta_{deviner}$ et θ_2, nous assignons donc θ_1 à $\theta_{deviner}$ et faisons une nouvelle hypothèse à mi-chemin entre les nouveaux θ_1 et θ_2. Sinon, si le signe de $\Delta x(\theta_{deviner})$ est identique au signe de $\Delta x(\theta_2)$, nous savons que la racine doit être comprise entre θ_1 et $\theta_{deviner}$. Nous affectons donc θ_2 à $\theta_{deviner}$ et faisons une nouvelle hypothèse à mi-chemin entre θ_1 et le nouveau θ_2. Nous continuons cette itération jusqu'à ce

que nous soyons *suffisamment proche*, c'est-à-dire $|\Delta x(\theta_{deviner})| < \epsilon$ pour une petite valeur de ϵ.

Pour le problème à résoudre, la cible doit être située à une distance x_{cible} et le point où le projectile touche le sol lorsqu'il est lancé à l'angle θ est $x_{sol}(\theta)$. Définir $\Delta x(\theta) = x_{sol}(\theta) - x_{cible}$ de telle sorte que $\Delta x(\theta) > 0$ si nous avons tiré trop loin et $\Delta x(\theta) < 0$ si nous avons tiré trop près. Ensuite, si $0 < x_{cible} < x_{max}$ où nous connaissons $x_{sol}(0°) = 0$ et $x_{sol}(45°) = x_{max}$, alors nous savons que $\theta_1 = 0°$ et $\theta_2 = 45°$ encadrent la racine. Le programme `tire.py` utilise la méthode de tir pour calculer la trajectoire d'un projectile lancé à partir de $x = 0$ avec une vitesse fixe et atterrissant au point $x = x_{sol}$. Le résultat de ce programme exécuté avec une vitesse initiale $v_0 = 10\ m\ s^{-1}$ et un emplacement cible $x_{cible} = 8\ m$ est présenté à la Fig. 6.5.

```
## NOM DU PROGRAMME: tire.py
#%% IMPORTATION
import numpy as np
import matplotlib.pyplot as plt
#%% CONSTANTES
# accélération de la pesanteur (m/s^2)
g = 9.8
# vitesse initiale (m/s)
v0 = 10.0
# plage cible (m)
xcible = 8.0
# comment nous devons nous rapprocher (m)
eps = 0.01
# pas de temps (s)
dt = 0.001
# angle (degrés) que le projectile tombe trop court
theta1 = 0.0
# angle (degrés) que le projectile tombe trop loin
theta2 = 45.0
# une valeur initiale> eps
dx = 2*eps
#%% BOUCLE PRINCIPALE: MÉTHODE DE TIRE
while abs(dx) > eps:
    # devinez à la valeur de thêta
    theta = (theta1+theta2)/2.0
    x = [0.0]
    y = [0.0]
    vx = [v0*np.cos(theta*np.pi/180.0)]
    vy = [v0*np.sin(theta*np.pi/180.0)]
    # MÉTHODE D'EULER
    i = 0
    while y[i] >= 0.0:
        # appliquer une différence finie approximative
        # aux équations du mouvement
        x += [x[i]+vx[i]*dt]
        y += [y[i]+vy[i]*dt]
        vx += [vx[i]]
        vy += [vy[i] - g*dt]
        i = i+1
    # nous avons touché le sol quelque part entre l'étape i-1 et i
    # interpoler pour trouver cet emplacement
    xsol = x[i - 1]+y[i - 1]*(x[i] - x[i - 1])/(y[i] - y[i - 1])
    # mettre à jour les limites encadrant la racine
    dx = xsol - xcible
    if dx < 0.0: # trop court: mettre à jour l'angle plus petit
        theta1 = theta
    else: # trop loin: mettre à jour un angle plus grand
        theta2 = theta
#%% GRAPHIQUE: TRAJECTOIRES
plt.plot(x, y, lw =2)
plt.plot([xcible], [0.0], 'o', ms=12)
plt.annotate('Cible', xy=(xcible, 0), xycoords='data', xytext=(5,5),
             textcoords='offset points')
plt.title(r"trajectoire d'un projectile avec $\theta$ = %.2f deg"% theta)
plt.xlabel('x [m]')
```

```
plt.ylabel('y [m]')
plt.ylim(ymin=0.0)
plt.grid()
# ENREGISTRER ET AFFICHER LA FIGURE
plt.savefig("tire.png")
plt.savefig("tire.pdf")
plt.show()
```

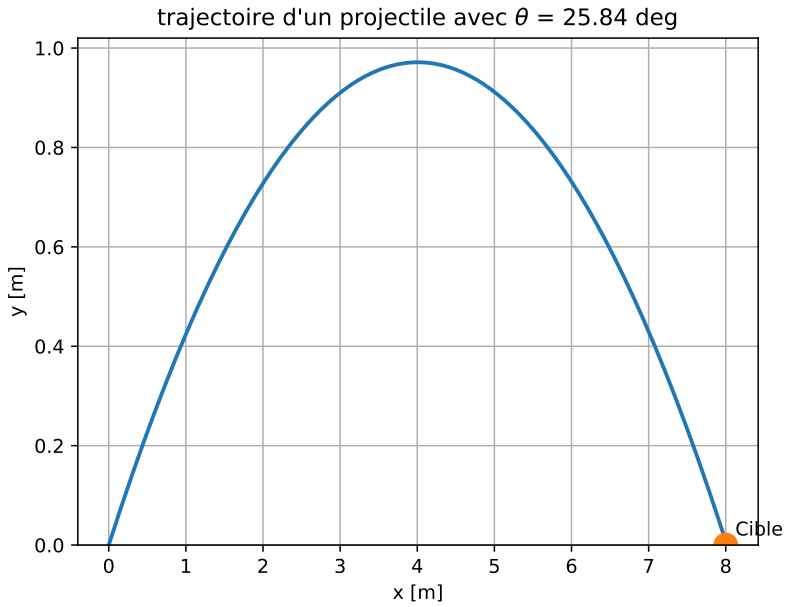


FIGURE 6.5 – Résultats de l'exécution du programme tire.py avec une vitesse initiale $v_0 = 10\ m\ s^{-1}$ et l'emplacement cible $x_{cible} = 8$ m. L'angle requis pour atteindre la cible est $\theta = 25.84°$.

6.4 Convergence et de stabilité de la méthode d'Euler : Cas des systèmes linéaires

En mécanique classique, les équations du mouvement d'un système mécanique (systèmes de points matériels, système de solides) sont des équations différentielles du second ordre par rapport au temps. La connaissance des positions et des vitesses des points à l'instant $t = 0$ suffit à déterminer le mouvement pour $t > 0$.

Ces équations sont souvent non linéaires car les forces elles-mêmes le sont (par exemple la force de gravitation) et car l'accélération est souvent une fonction non linéaire des degrés de liberté. Dans ce cas, il est fréquent que l'on ne connaisse pas de solution analytique exacte. On est alors amené à rechercher une solution approchée par une méthode numérique.

Cette partie du cours explique le principe de ce type d'intégration numérique. On prendra l'exemple de l'oscillateur harmonique (dont la solution exacte est connue) auquel on appliquera la méthode numérique d'Euler. On abordera les notions importantes de *convergence* et de *stabilité*.

On verra aussi des variantes de la méthode d'Euler, qui peuvent être utilisées pour résoudre des systèmes conservatifs à N corps, par exemple en dynamique moléculaire.

De manière générale soit le système d'équations différentielles suivant :

$$\dot{\boldsymbol{u}} = f(\boldsymbol{u}) \tag{6.11}$$

Où $\boldsymbol{u}$ peut être un vecteur d'état et $f(\boldsymbol{u})$ peut être linéaire ou non linéaire.

Soit $f(\boldsymbol{u}) = \boldsymbol{A} \cdot \boldsymbol{u}$ avec $\boldsymbol{A}$ une matrice. Donc on peut écrire l'équation (6.4) comme suit :

$$\dot{\boldsymbol{u}} = \boldsymbol{A} \cdot \boldsymbol{u} \quad avec\ \boldsymbol{u}(t=0) = \boldsymbol{u}_0 \tag{6.12}$$

La solution analytique exacte d'un tel système est de la forme :

$$\boldsymbol{u}(t) = e^{\boldsymbol{A}t} \cdot \boldsymbol{u}_0 \tag{6.13}$$

On se propose d'appliquer différentes méthodes d'Euler au système (6.4).

6.4.1 La méthode d'Euler explicite (progressive)

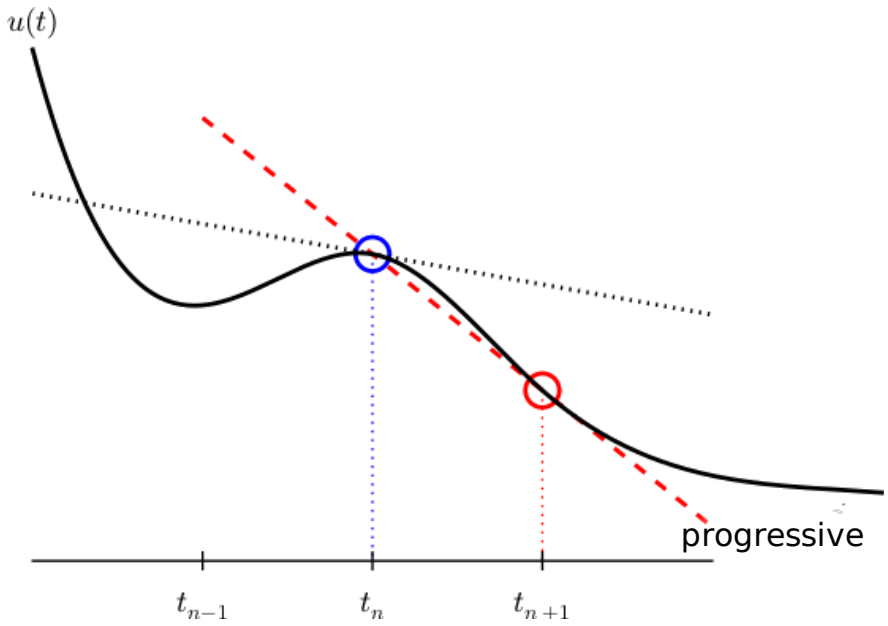


FIGURE 6.6 – Illustration d'une approximation par différence progressive de la dérivée.

$$\frac{\boldsymbol{u}_{k+1} - \boldsymbol{u}_k}{\Delta t} \approx \dot{\boldsymbol{u}}_k = f(\boldsymbol{u}_k) \tag{6.14}$$

$$\boldsymbol{u}_{k+1} = \boldsymbol{u}_k + \Delta t f(\boldsymbol{u}_k) \tag{6.15}$$

Si $\dot{\boldsymbol{u}} = \boldsymbol{A} \cdot \boldsymbol{u}$ alors ;

$$\boldsymbol{u}_{k+1} = \boldsymbol{u}_k + \Delta t \boldsymbol{A} \cdot \boldsymbol{u}_k = (\boldsymbol{I} + \Delta t \boldsymbol{A}) \cdot \boldsymbol{u}_k \tag{6.16}$$

Où $\boldsymbol{I}$ est la matrice identité.

6.4.2 La méthode d'Euler implicite (rétrograde)

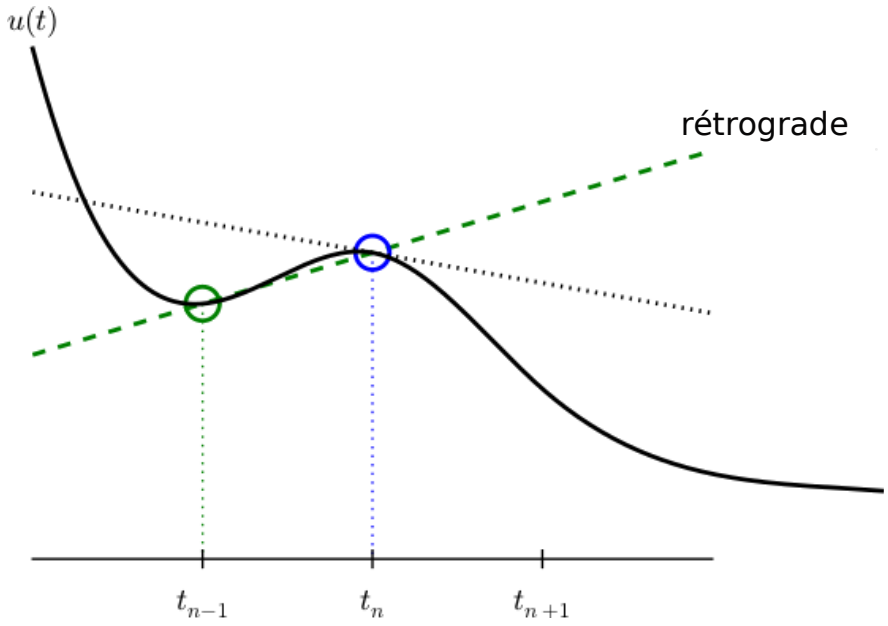


FIGURE 6.7 – Illustration d'une approximation par différence rétrograde de la dérivée.

$$\frac{\boldsymbol{u}_{k+1} - \boldsymbol{u}_k}{\Delta t} \approx \dot{\boldsymbol{u}}_{k+1} = f(\boldsymbol{u}_{k+1}) \tag{6.17}$$

$$\boldsymbol{u}_{k+1} = \boldsymbol{u}_k + \Delta t f(\boldsymbol{u}_{k+1}) \tag{6.18}$$

Si $\dot{\boldsymbol{u}} = \boldsymbol{A} \cdot \boldsymbol{u}$ alors ;

$$\boldsymbol{u}_{k+1} = \boldsymbol{u}_k + \Delta t \boldsymbol{A} \cdot \boldsymbol{u}_{k+1} \tag{6.19}$$

$$(\boldsymbol{I} - \Delta t \boldsymbol{A}) \cdot \boldsymbol{u}_{k+1} = \boldsymbol{u}_k \tag{6.20}$$

$$\boldsymbol{u}_{k+1} = (\boldsymbol{I} - \Delta t \boldsymbol{A})^{-1} \cdot \boldsymbol{u}_k \tag{6.21}$$

Où $\boldsymbol{I}$ est la matrice identité.

6.4.3 Exemple : Oscillateur libre amorti [masse, ressort, amortisseur]

Un bloc de masse m est lié à l'extrémité libre d'un ressort de raideur k , de longueur au repos l, de masse négligeable et d'élasticité parfaite, l'autre extrémité du ressort étant fixe. Le système est supposé dans l'espace (on néglige la force de pesanteur). Le seul mouvement possible pour le bloc est une translation suivant x ; on assimilera le bloc à un point matériel M.

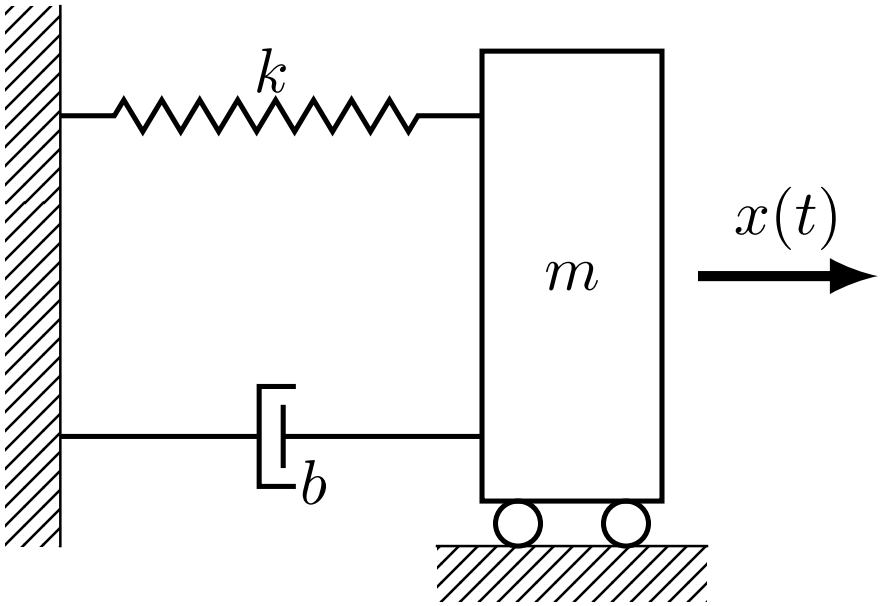


FIGURE 6.8 – Schéma d'un système dynamique oscillant amorti unidimensionnel.

Bilan des forces.

- La force de rappel $\overrightarrow{F_r} = -k\overrightarrow{x}$ où k est un coefficient positif et $\overrightarrow{x}$ le vecteur position de M.
- Le système est amorti. L'amortissement de type visqueux est représenté par un amortisseur qui exerce la force dissipative (ou force d'amortissement visqueux) $\overrightarrow{F_a} = -b\overrightarrow{v}$ où b est un coefficient positif et $\overrightarrow{v}$ le vecteur vitesse de M.

Équation de mouvement. La deuxième loi de Newton pour le système peut être écrite avec l'accélération multipliée par la masse du côté gauche et la somme des forces du côté droit :

$$m\overrightarrow{a} = \overrightarrow{F_a} + \overrightarrow{F_r} \tag{6.22}$$

$$m\ddot{x} = -b\dot{x} - kx \tag{6.23}$$

$$m\ddot{x} + b\dot{x} + kx = 0 \tag{6.24}$$

On réécrit cette équation sous la forme canonique suivante :

$$\ddot{x} + 2\zeta\omega_0\dot{x} + \omega_0^2 x = 0 \tag{6.25}$$

avec $\omega_0 = \sqrt{\frac{k}{m}}$ désigne une *pulsation caractéristique* et $\zeta = \frac{b}{2\sqrt{km}}$ est une quantité positive sans dimension, appelée *taux d'amortissement*.

C'est une équation différentielle linéaire d'ordre 2 à coefficients constants.

On peut trouver numériquement la solution de l'équation (6.25) à l'aide des méthodes d'Euler à partir du système d'équations différentielles ordinaires suivant :

$$\dot{x} = v \tag{6.26}$$
$$\dot{v} = -2\zeta\omega_0 v - \omega_0^2 x \tag{6.27}$$
$$\tag{6.28}$$

$$\frac{d}{dt}\begin{pmatrix} x \\ v \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -\omega_0^2 & -2\zeta\omega_0 \end{pmatrix} \cdot \begin{pmatrix} x \\ v \end{pmatrix} \tag{6.29}$$

L'équation (6.4.3) est de la forme : $\dot{\boldsymbol{u}} = \boldsymbol{A} \cdot \boldsymbol{u}$ avec :

$$\boldsymbol{A} = \begin{pmatrix} 0 & 1 \\ -\omega_0^2 & -2\zeta\omega_0 \end{pmatrix}$$

et

$$\boldsymbol{u} = \begin{pmatrix} x \\ v \end{pmatrix}$$

Supposons que nous voulions résoudre le problème avec : $\omega_0 = 2\pi$, $\zeta = 0.25$, $\boldsymbol{u_0} = \begin{pmatrix} x(t=0) \\ v(t=0) \end{pmatrix} = \begin{pmatrix} 2 \\ 0 \end{pmatrix}$, $\Delta t = 0.01$ pour $t \in [0, 10]$. Ce sera une solution sinusoïdale amortie.

Solution avec la méthode d'Euler explicite. Nous implémentons l'expression explicite d'Euler montrée dans (6.16) dans le code python suivant :

```
## NOM DU PROGRAMME: OscillateurEulerExp.py
#% IMPORTATION
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d.axes3d import Axes3D
# SYSTÈME: OSCILLATEUR LIBRE AMORTI
w = 2*np.pi # fréquence propre
a = 0.25    # rapport d'amotissement
```

```
A = np.array([[0, 1], [-w**2, -2*a*w]])
dt = 0.01 # pas du temps
Tf = 10 # temps finale de la simulation
nsteps = int(Tf/dt)
# CONDITION INITIAL: à t = 0; x = 2, v = 0
u0 = np.array([2,0])
#%% ITÉRATION: EULER ExPLICITE
Texp = np.zeros(nsteps)
Uexp = np.zeros((2, nsteps))
Texp[0] = 0.0
Uexp[:,0] = u0
for k in range(nsteps-1):
    Texp[k+1] = Texp[k] + dt
    Uexp[:,k+1] = np.dot((np.eye(2) + dt * A), Uexp[:,k])

plt.figure(figsize=(10,5))
# PLOT POSITION vs TEMPS
plt.suptitle("Simulation d'un oscillateur libre amorti avec un pas d'intégration "+ r"$ \Delta t= %.2f$"%dt,
             fontweight = "bold")
plt.subplot(1,2,1)
plt.plot(Texp,Uexp[0,:], linewidth=2, color ='k')
plt.xlabel("Temps")
plt.ylabel("Position")
plt.title("Trajectoire de la mass M (Euler explicite)")
# DIAGRAMME DE PHASE 2D
plt.subplot(1,2,2)
plt.plot(Uexp[0,:],Uexp[1,:], linewidth=2, color ='k')
plt.xlabel("Position")
plt.ylabel("Vitesse")
plt.title("Trajectoire de phase (Euler explicite)")
plt.savefig("EulerExp1D.png"); plt.savefig("EulerExp1D.pdf")
# DIAGRAMME DE PHASE 3D
plt.figure()
ax = plt.axes(projection="3d")
ax.plot(Texp, Uexp[0,:],Uexp[1,:], linewidth=2, color ='k')
ax.set_xlabel("Temps")
ax.set_ylabel("Position")
ax.set_zlabel("Vitesse")
ax.set_title("Trajectoire de phase (Euler explicite)")
plt.savefig("EulerExp3D.png"); plt.savefig("EulerExp3D.pdf")
plt.show()
```

La figure 6.9 est générée par le code `OscillateurEulerExp.py`, montrant la divergence et l'instabilité de la méthode Euler explicite. En effet, le pas d'intégration Δt agit considérablement sur la qualité de la simulation et donne un résultat inacceptable physiquement.

Dans le cas d'intégration avec la méthode d'Euler explicite, la figure 6.10 montre que nous avons un problème d'augmentation d'amplitude dans le cas d'un oscillateur non amorti (courbe bleue pour $\zeta = 0$). Plus le temps de simulation est long, plus l'amplitude augmente, ce qui n'est pas ce que nous attendons de l'évolution du système dans le temps. En d'autres termes, l'amplitude devrait être constante dans le temps pour un système oscillant non amorti.

Solution avec la méthode d'Euler implicite. Nous implémentons l'expression implicite d'Euler montrée dans (6.21) dans le code python suivant :

```
## NOM DU PROGRAMME: OscillateurEulerImp.py
#% IMPORTATION
import numpy as np
from numpy.linalg import inv
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d.axes3d import Axes3D
# SYSTÈME: OSCILLATEUR LIBRE AMORTI
w = 2*np.pi # fréquence propre
a = 0.25    # rapport d'amotissement
```

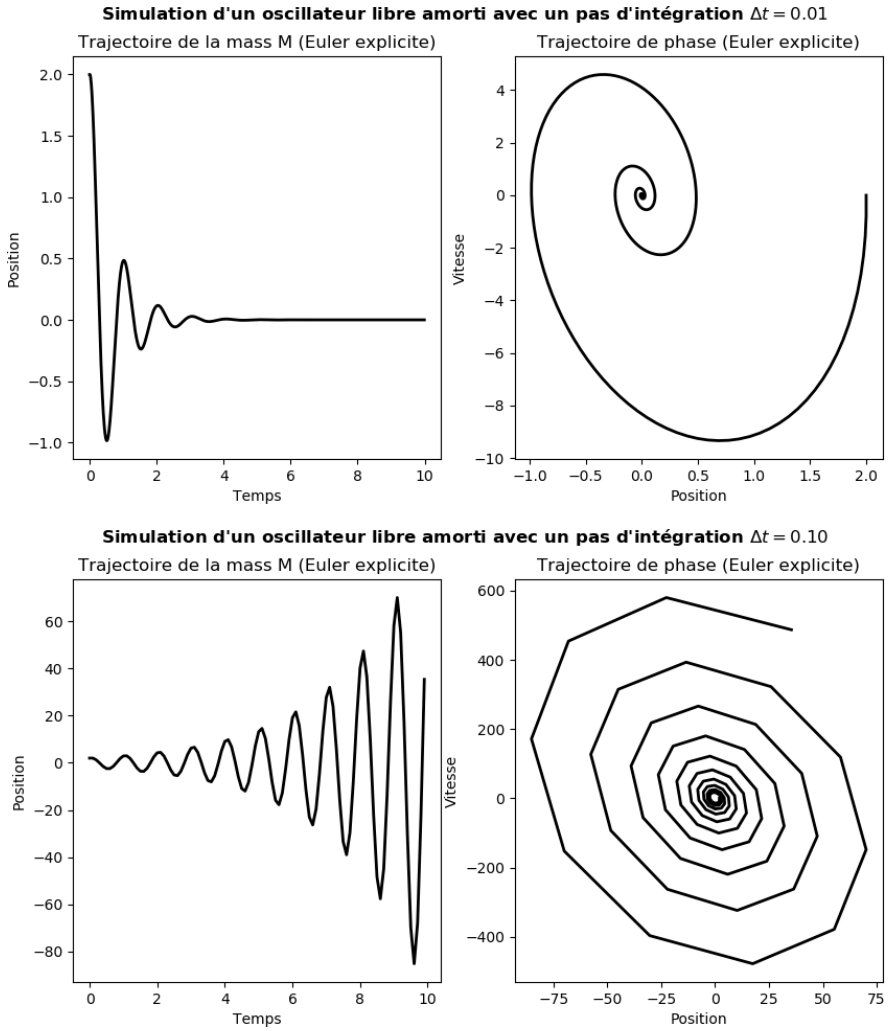


FIGURE 6.9 – Simulation d'un système oscillant avec différents pas de temps ; $\Delta t = 0.01$ et $\Delta t = 0.1$ et pour $\zeta = 0.25$.

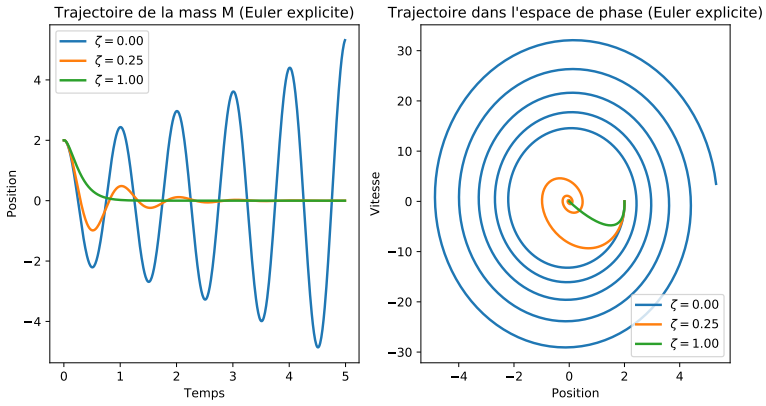


FIGURE 6.10 – Simulation d'un système oscillant avec différentes valeurs de ζ et pour $\Delta t = 0.01$.

```
A = np.array([[0, 1], [-w**2, -2*a*w]])
dt = 0.1 # pas du temps
Tf = 10 # temps finale de la simulation
nsteps = int(Tf/dt)
# CONDITION INITIAL: à t = 0; x = 2, v = 0
u0 = np.array([2,0])
#%% ITÉRATION: EULER IMPLICITE
Timp = np.zeros(nsteps)
Uimp = np.zeros((2, nsteps))
Timp[0] = 0.0
Uimp[:,0] = u0
for k in range(nsteps-1):
    Timp[k+1] = Timp[k] + dt
    Uimp[:,k+1] = np.dot(inv(np.eye(2) - dt * A), Uimp[:,k])

plt.figure(figsize=(10,5))
# PLOT POSITION vs TEMPS
plt.suptitle("Simulation d'un oscillateur libre amorti avec un pas d'intégration "+ r"$ \Delta t= %.2f$"%dt,
             fontweight = "bold")
plt.subplot(1,2,1)
plt.plot(Timp,Uimp[0,:], linewidth=2, color ='k')
plt.xlabel("Temps")
plt.ylabel("Position")
plt.title("Trajectoire de la mass M (Euler implicite)")
# DIAGRAMME DE PHASE 2D
plt.subplot(1,2,2)
plt.plot(Uimp[0,:],Uimp[1,:], linewidth=2, color ='k')
plt.xlabel("Position")
plt.ylabel("Vitesse")
plt.title("Trajectoire de phase (Euler implicite)")
plt.savefig("Eulerimp1D_2.png"); plt.savefig("Eulerimp1D_2.pdf")
# DIAGRAMME DE PHASE 3D
plt.figure()
ax = plt.axes(projection="3d")
ax.plot(Timp, Uimp[0,:],Uimp[1,:], linewidth=2, color ='k')
ax.set_xlabel("Temps")
ax.set_ylabel("Position")
ax.set_zlabel("Vitesse")
ax.set_title("Trajectoire de phase (Euler implicite)")
plt.savefig("Eulerimp3D_2.png"); plt.savefig("Eulerimp3D_2.pdf")
plt.show()
```

La figure 6.11 est générée par le code `OscillateurEulerImp.py`, montrant que la méthode d'Euler implicite est plus stable que la méthode Euler explicite. Nous remarquons toujours qu' il y a un effet du changement du pas d'intégration Δt sur la qualité de la simulation mais le résultat du calcul est désormais acceptable physiquement.

Même problème avec l'amplitude pour le cas d'intégration avec la méthode implicite d'Euler, la figure 6.12 montre que nous avons un problème de diminution d'amplitude dans le cas d'un oscillateur non amorti (courbe bleue pour $\zeta = 0$). Comme indiqué ci-dessus, l'amplitude devrait être constante dans le temps pour un système oscillant non amorti.

6.4.4 Conclusion

La conclusion ici est que la méthode Euler implicite est plus stable que celle explicite. Les deux méthodes posent un problème fondamental avec ses amplitudes croissantes et décroissantes, pour le cas d'oscillateur libre non amorti, et qu'un très petit Δt est nécessaire pour obtenir des résultats satisfaisants. Plus la simulation est longue, plus Δt doit être petit. Il est certainement temps de rechercher des méthodes numériques plus stables et plus efficaces tels que les méthodes de Runge-Kutta.

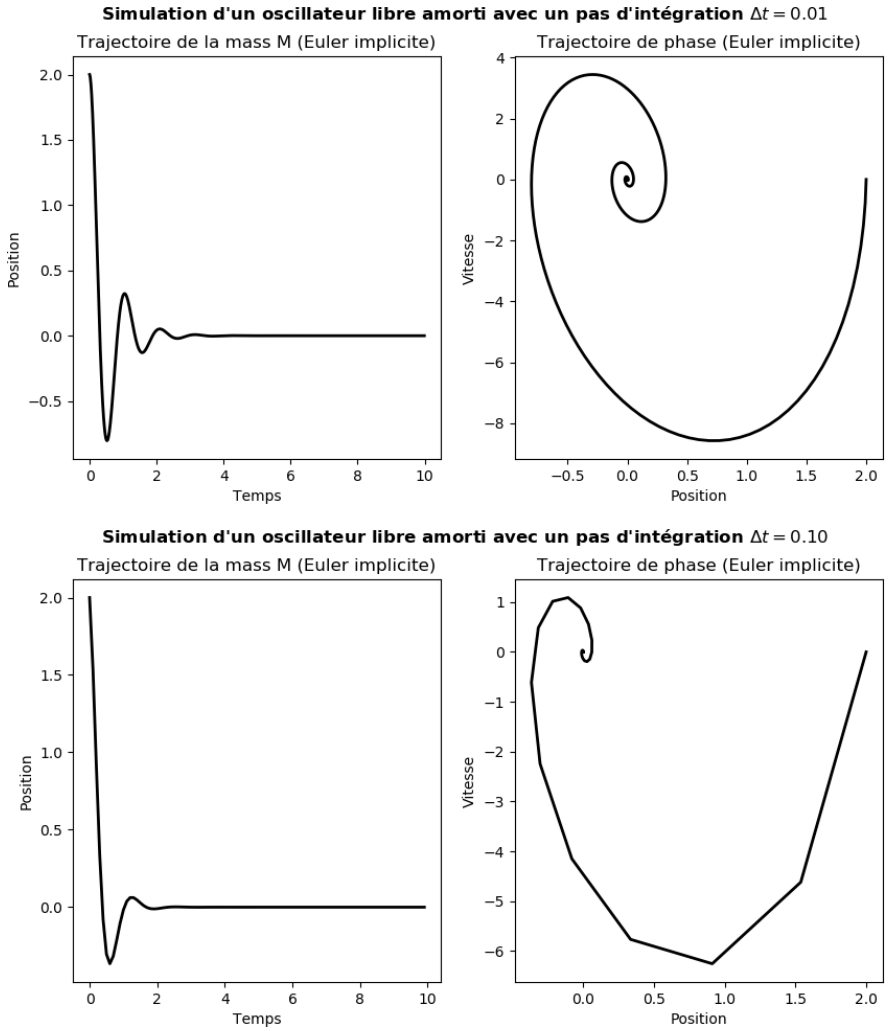


FIGURE 6.11 – Simulation d'un système oscillant avec différents pas de temps ; $\Delta t = 0.01$ et $\Delta t = 0.1$ et pour $\zeta = 0.25$.

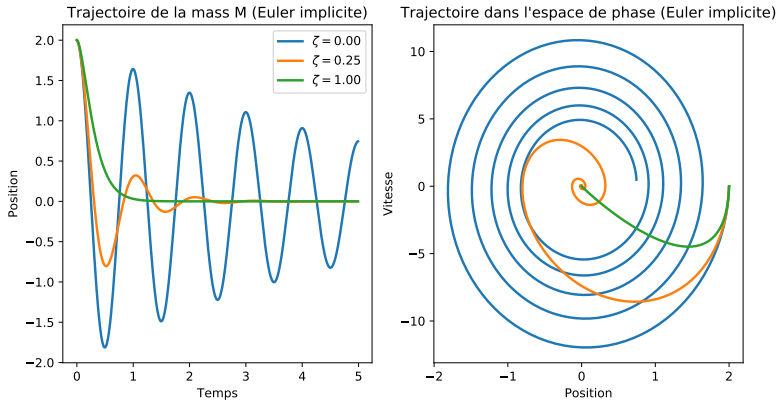


FIGURE 6.12 – Simulation d'un système oscillant avec différentes valeurs de ζ et pour $\Delta t = 0.01$.

6.5 La méthode de Runge-Kutta d'ordre 4

Les méthodes de Runge-Kutta (ou RK), l'ordre 2 ou 4, sont très couramment utilisées pour la résolution d'équations différentielles ordinaires (EDO). Ce sont des méthodes à pas unique, directement dérivées de la méthode d'Euler, qui est une méthode RK1.

Elles ont l'avantage d'être simples à programmer et assez stables pour les fonctions courantes de la physique. Sur le plan de l'analyse numérique, elles ont surtout l'immense avantage de ne pas nécessiter autre chose que la connaissance des valeurs initiales.

6.5.1 Algorithme de Runge-Kutta d'ordre 4

On part de la formule d'Euler sous sa forme scalaire, qui donne : $y_{n+1} = y_n + h * f(x_n, y_n), et x_{n+1} = x_n + h$.

La méthode RK du deuxième ordre produit deux coefficients k_1 et k_2, qui permettent d'écrire :

$$\begin{aligned} k_1 &= h * f(x_n, y_n) \\ k_2 &= h * f(x_n + h/2, y_n + k_1/2) \\ y_{n+1} &= y_n + k_2 + O(h^3) \end{aligned}$$

Cette méthode exige donc deux évaluations de f. L'erreur de consistance est en $O(h^3)$ et l'erreur globale de convergence est d'ordre $O(h^2)$. Pour obtenir plus de précision, mais en doublant le temps de calcul puisqu'on procède à 4 évaluations de f, voici la méthode RK4 :

$$\begin{aligned} k_1 &= h * f(x_n, y_n) \\ k_2 &= h * f(x_n + h/2, y_n + k_1/2) \\ k_3 &= h * f(x_n + h/2, y_n + k_2/2) \\ k4 &= h * f(x_n + h, y_n + k_3) \\ y_{n+1} &= y_n + k_1/6 + k_2/3 + k_3/3 + k_4/6 + O(h^5) \end{aligned}$$

6.5.2 Exemple : Système dynamique différentiel de Lorenz (attracteur de Lorenz)

L'attracteur de Lorenz, connu aussi sous le nom de *"papillon de Lorenz"*, est sans doute le plus connu des systèmes dynamiques non linéaires, essentiellement pour son aspect esthétique !

Le problème avec la prédiction météorologique est l'interdépendance de tous les paramètres de l'atmosphère. Pour prendre un exemple, lorsque vous avez une zone de dépression, il y a un déplacement d'une masse d'air voisine. C'est ici une réaction thermodynamique, l'atmosphère s'organise pour que son enthalpie soit minimum alors qu'en même temps, le sol continue à chauffer l'air créant de nouveau un déséquilibre. De même, l'ensoleillement n'étant pas constant à la surface de la Terre (dépendant de la situation géographique et des saisons), nous avons donc formation de diverses cellules de convections qui se retrouvent interdépendantes. Une perturbation sur l'une de ces cellules peut être retrouvée sous la forme d'une autre

perturbation sur une autre cellule de convection. C'est ici l'idée qui s'est dégagée de la remarque de Lorenz : *"Le battement d'ailes d'un papillon au Brésil peut-il provoquer une tornade au Texas ?"*

Lorsque Lorenz a découvert cette théorie en 1963, il étudiait la météorologie au MIT. Comme nous l'avons dit, il fut l'un des premiers a avoir pu utiliser un ordinateur ! Bien sûr, ce n'était pas du tout les mêmes machines que nous utilisons aujourd'hui ! Ces derniers occupaient une salle entière, chauffaient comme un diable et avaient une capacité de calcul inférieure aux premiers smartphones commercialisés. Il fallait donc être très patient lorsque vous vouliez faire une modélisation et chaque erreur faisait perdre du temps à vos collègues (car à cette époque, il n'y avait que quelques ordinateurs au MIT) et gaspillait énormément d'énergie.

Le système différentiel résulte d'une simplification assez drastique de l'ensemble des équations différentielles en jeu. C'est un système paramétrique dont voici l'expression :

$$\begin{aligned}
\frac{dx}{dt} &= \sigma\,[y(t) - x(t)] \\
\frac{dy}{dt} &= \rho\ x(t) - y(t) - x(t)\ z(t) \\
\frac{dz}{dt} &= x(t)\ y(t) - \beta\ z(t)
\end{aligned}$$

σ, β et ρ sont trois paramètres strictement positifs, fixés.

- σ dépend des propriétés du fluide (c'est la constante de Prandtl , qui caractérise la viscosité et la conductivité thermique du fluide),
- β varie avec la géométrie de la cellule de convection,
- ρ varie en fonction du gradient de température dans la cellule.

Les variables dynamiques x, y et z représentent l'état du système à chaque instant :

- x caractérise le taux de convection,
- y et z le gradient de température, respectivement horizontal et vertical.

A vrai dire, la signification réelle des paramètres et des variables importe peu pour l'étude de la dynamique du système de Lorenz. Il faut simplement savoir que l'on fixe généralement $\sigma = 10$ et $\beta = 8/3$ (les paramètres physiques et géométriques) et que l'on étudie le comportement en fonction de la variation de ρ et des conditions initiales.

Le tracé de la trajectoire dans l'espace de phase est obtenu avec le programme `AttracteurLorenzRK4.py`, qui est écrit selon la méthode RK4 et avec les valeurs de paramètres $\sigma = 10$, $\beta = 8/3$ et $\rho = 28$.
Les conditions initiales pour [x0,y0,z0] sont [1,1,20].

```
## NOM DU PROGRAMME: AttracteurLorenzRK4.py
#% IMPORTATION
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
#Initial values
sigma = 10
beta = 8.0/3.0
rho = 28.0
#Compute the time-derivative of a Lorenz system.
def xt(x, y, z, t):
    return (sigma*(y - x))
def yt(x, y, z, t):
    return (rho*x - y - x*z)
def zt(x, y, z, t):
    return (-1*(beta*z) + x*y)
# définition de la fonction RK4
def RungeKutta4(xt,yt,zt,n = 3500, tf = 30):
    x = np.zeros(n)
    y = np.zeros(n)
    z = np.zeros(n)
    t = np.zeros(n)
    x[0],y[0], z[0], t[0] = 1.0, 1.0, 20, 0
    dt = tf/float(n)
    # calculer la solution approximative RK4.
    for k in range (n-1):
        t[k+1] = t[k] + dt
        k1 = xt(x[k], y[k], z[k], t[k])
        l1 = yt(x[k], y[k], z[k], t[k])
        m1 = zt(x[k], y[k], z[k], t[k])

        k2 = xt((x[k] + 0.5*k1*dt), (y[k] + 0.5*l1*dt), \
                (z[k] + 0.5*m1*dt), (t[k] + dt/2))
        l2 = yt((x[k] + 0.5*k1*dt), (y[k] + 0.5*l1*dt), \
                (z[k] + 0.5*m1*dt), (t[k] + dt/2))
        m2 = zt((x[k] + 0.5*k1*dt), (y[k] + 0.5*l1*dt), \
                (z[k] + 0.5*m1*dt), (t[k] + dt/2))

        k3 = xt((x[k] + 0.5*k2*dt), (y[k] + 0.5*l2*dt), \
                (z[k] + 0.5*m2*dt), (t[k] + dt/2))
        l3 = yt((x[k] + 0.5*k2*dt), (y[k] + 0.5*l2*dt), \
                (z[k] + 0.5*m2*dt), (t[k] + dt/2))
        m3 = zt((x[k] + 0.5*k2*dt), (y[k] + 0.5*l2*dt), \
                (z[k] + 0.5*m2*dt), (t[k] + dt/2))

        k4 = xt((x[k] + k3*dt), (y[k] + l3*dt), (z[k] + m3*dt), (t[k] + dt))
        l4 = yt((x[k] + k3*dt), (y[k] + l3*dt), (z[k] + m3*dt), (t[k] + dt))
        m4 = zt((x[k] + k3*dt), (y[k] + l3*dt), (z[k] + m3*dt), (t[k] + dt))

        x[k+1] = x[k] + (dt*(k1 + 2*k2 + 2*k3 + k4) / 6)
        y[k+1] = y[k] + (dt*(l1 + 2*l2 + 2*l3 + l4) / 6)
        z[k+1] = z[k] + (dt*(m1 + 2*m2 + 2*m3 + m4) / 6)
    return x, y, z, t

x, y, z, t = RungeKutta4(xt,yt,zt)

plt.figure (figsize = (8,5))
plt.plot ( t, x, linewidth = 1, color = 'b' )
plt.plot ( t, y, linewidth = 1, color = 'r' )
plt.plot ( t, z, linewidth = 1, color = 'g' )
plt.xlabel ( 'Temps' )
plt.ylabel ( 'x(t), y(t), z(t)' )
plt.title ('Évolution des coordonnées x, y et z en fonction du temps' )
plt.legend(['x(t)','y(t)','z(t)'], loc = 2)
plt.savefig ('lorenz_ode_components.png'); plt.savefig ('lorenz_ode_components.pdf' )
plt.show ( )
# FIGURE 3d: ATTRACTEUR DE LORENZ
```

```
fig = plt.figure()
ax = plt.axes(projection = '3d' )
ax.plot ( x, y, z, linewidth = 0.5, color = 'k' )
ax.set_xlabel ('x(t)')
ax.set_ylabel ('y(t)')
ax.set_zlabel ('z(t)')
ax.set_title ('Attracteur de Lorenz pour ' + r"$\rho = %.2f$"%rho)
plt.tight_layout()
plt.savefig ('lorenz_ode_3d.png'); plt.savefig ('lorenz_ode_3d.pdf')
plt.show()
```

Voilà ce que cela donne avec le programme `AttracteurLorenzRK4.py` :

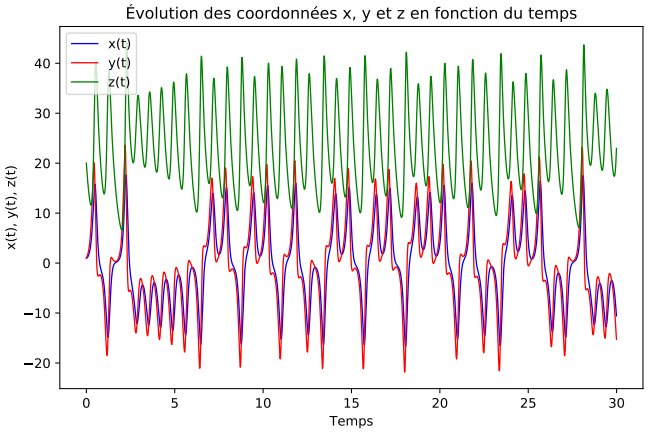


FIGURE 6.13 – Évolution des coordonnées x, y et z en fonction du temps. $\sigma = 10$, $\beta = 8/3$ et $\rho = 28$. Les conditions initiales sont pour [x0,y0,z0] sont [1,1,20].

Lorsque nous modélisons ce système dans l'espace des phases, nous avons apparition d'une structure fractale que nous appelons "Attracteur de Lorenz". C'est cette construction mathématique qui a probablement donné l'idée du papillon à Lorenz pour illustrer sa mécanique Chaotique lors de sa conférence de 1963 !

Nous remarquons que notre système va devenir chaotique lorsque ρ va dépasser une valeur critique $\rho_c = 19.44$

Lorsque notre système n'est pas chaotique, nous avons la formation d'un seul attracteur, cependant, lorsque ρ dépasse la valeur ρ_c, notre papillon déploie ses ailes pour nous inviter dans le monde du Chaos ! Nous pouvons alors voir la présence d'un deuxième attracteur venant perturber la trajectoire de notre système qui rester à naviguer entre les deux puits de gravité de ces deux attracteurs. On qualifie également cette figure "d'attracteur étrange" car les trajectoires ne se coupent pas et semblent évoluer au hasard.

6.6 Travaux dirigés

6.6.1 Exercice 1 : Pendule simple

On considère un pendule simple de masse $m = 1\ kg$, de longueur $l = 1\ m$ qui va osciller d'arrière en avant à cause du champ de gravité de la Terre $g = 9.8\ m/s^2$.

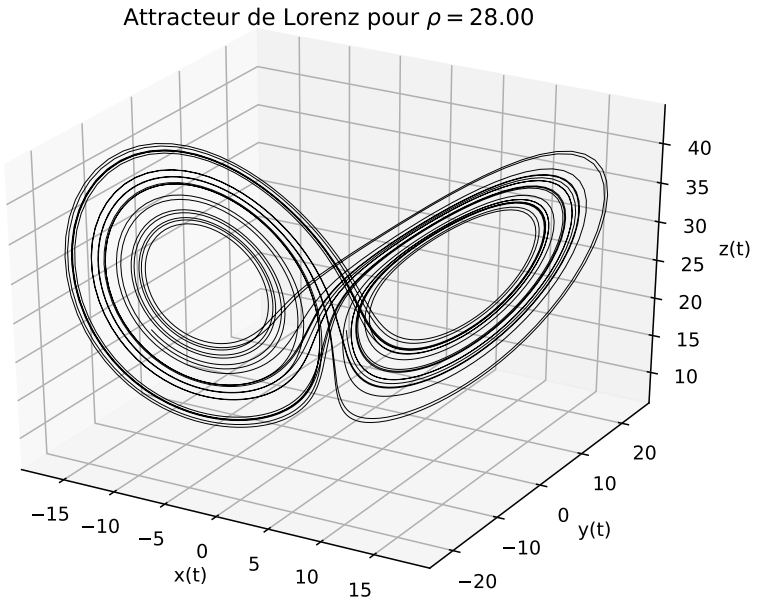


FIGURE 6.14 – Attracteur de Lorenz pour $\sigma = 10$, $\beta = 8/3$ et $\rho = 28$. Les conditions initiales sont pour [x0,y0,z0] sont [1,1,20].

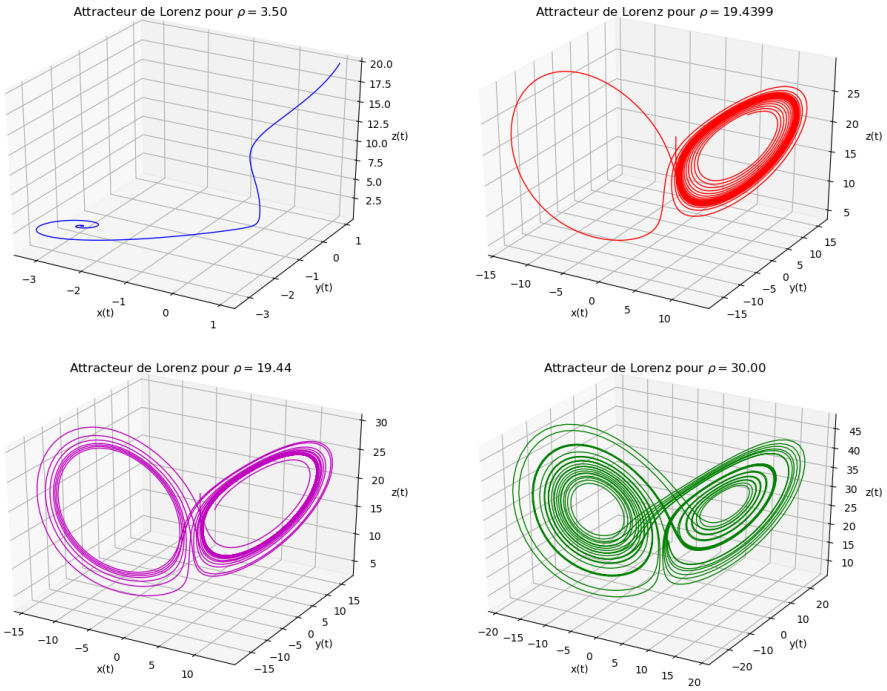
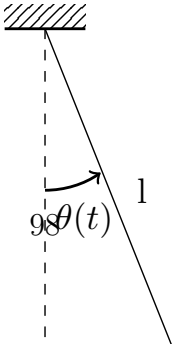


FIGURE 6.15 – Formation d'un deuxième attracteur lorsque ρ dépasse la valeur critique $\rho_c = 19.44$.



Le pendule a l'équation du mouvement :

$$\ddot{\theta} = -\frac{g}{l} sin(\theta) \tag{6.30}$$

Pour les petites amplitudes d'oscillation, $\theta \ll 1$, on peut faire l'approximation $sin(\theta) \approx \theta$, on retrouve alors l'équation différentielle d'un oscillateur harmonique :

$$\ddot{\theta} = -\frac{g}{l}\theta \tag{6.31}$$

La solution exacte de cette équation est simplement :

$$\theta(t) = \theta_0 \; cos(\omega_0 t) \tag{6.32}$$

où $\omega_0 = \sqrt{g/l}$ et nous avons supposé que le pendule partait du repos avec un déplacement initial $\theta_0 = 0.2 \; rad$.

Nous allons transformer l'équation différentielle d'ordre 2 (Eq. (6.31)) en deux équations différentielles d'ordre 1 afin de pouvoir utiliser simplement la méthode d'Euler. En posant $\omega(t) = \dot{\theta}(t)$ la vitesse angulaire du pendule, on obtient le système de deux fonctions inconnues suivant :

$$\dot{\theta}(t) = \omega(t) \tag{6.33}$$

$$\dot{\omega}(t) = -\omega_0^2 \; \theta(t) \tag{6.34}$$

Pour résoudre ce système nous devons connaître les deux conditions initiales suivantes :

$$\theta(0) = \theta_0$$
$$\omega(0) = 0$$

a) Définir une fonction `sol_exacte(t)` qui renvoie la solution exacte de l'oscillateur harmonique donnée par l'équation (6.32). Tracer cette solution pour $t \in [0, 10]$ et pour un pas de $\Delta t = 0.01$ s.

Indication.

- Utiliser la fonction `numpy.arange()` pour créer le vecteur temps `t`.
- Utiliser la fonction `matplotlib.pyplot.plot()` pour tracer `sol_exacte(t)`.

b) Rappeler l'expression de la méthode *d'Euler explicite* pour ce système.

c) Calculer $\boldsymbol{u} = \begin{pmatrix} \theta(t) \\ \omega(t) \end{pmatrix}$ avec la méthode d'Euler explicite pour $t \in [0, 10]$ et pour un pas d'intégration $\Delta t = 0.01$ s.

Tracer :

- Dans un même graphique, la variation de l'amplitude d'oscillation θ en fonction du temps t et le diagramme des phases (vitesse angulaire ω en fonction de θ).
- Dans un graphique 3D, la vitesse angulaire ω et l'amplitude d'oscillation θ en fonction du temps t.

Que remarquez-vous pour le résultat trouvé ?

Indication. On vous donne les instructions nécessaires pour reproduire un graphique en 3D :

```
from mpl_toolkits.mplot3d.axes3d import Axes3D
plt.figure()
ax = plt.axes(projection="3d")
ax.plot(....)
```

d) Rappeler l'expression de la méthode *d'Euler implicite* pour ce système.

e) Calculer $\boldsymbol{u} = \begin{pmatrix} \theta(t) \\ \omega(t) \end{pmatrix}$ avec la méthode d'Euler implicite pour $t \in [0, 10]$ et pour un pas d'integration $\Delta t = 0.01$ s.

Tracer :

- Dans un même graphique, la variation de l'amplitude d'oscillation θ en fonction du temps t et le diagramme des phases (vitesse angulaire ω en fonction de θ).
- Dans un graphique 3D, la vitesse angulaire ω et l'amplitude d'oscillation θ en fonction du temps t.

Que remarquez-vous pour le résultat trouvé ?

f) Tracer dans un même graphique pour $t \in [0, 10]$ et avec un pas $\Delta t = 0.01$ s :

- `sol_exacte(t)` calculée dans **a)**.
- $\theta(t)$ calculée dans **c)** par la méthode d'Euler explicite.
- $\theta(t)$ calculée dans **e)** par la méthode d'Euler implicite.

Que remarquez-vous si nous modifions la valeur du pas d'intégration par $\Delta\ t =\ 0.001$ s ? Expliquer le résultat trouvé.

6.6.2 Exercice 2 : Comparaison des schémas d'Euler explicite et implicite

On considère le problème de Cauchy :

$$\frac{dz(t)}{dt} = 1 - \frac{t}{\mu},\ t \in \Re,\ z(0) = z_0 \tag{6.35}$$

On rappelle que la solution exacte de ce problème est donnée par :

$$z(t) = \mu - (\mu - z_0)e^{-\frac{t}{\mu}} \tag{6.36}$$

a) Définir une fonction `sol_exacte(t, mu, z0)` qui renvoie la solution exacte donnée par l'équation (6.36). Tracer sur un même graphique pour $\mu = 1$ et $z_0 \in \{0, 1, 2\}$ ces solutions. Soit $t \in [0, 2]$ et pour un pas de $\Delta t = 0.1$ s.

b) Même questions pour $\mu = 0.05$ et $z_0 \in \{0, 1, 2\}$.

On suppose dans cette question que $\mu = 0.05$ et que $z_0 = 2$.

c) Rappeler l'expression de la méthode *d'Euler explicite* pour ce problème. Calculer $z(t)$ avec la méthode *d'Euler explicite* pour $t \in [0,2]$ et pour un pas d'intégration $\Delta t = 0.1$ s.

d) Montrer que l'expression de la méthode *d'Euler implicite* est :

$$z_{n+1} = \frac{z_n + \Delta t}{1 + \frac{\Delta t}{\mu}},\ n = 0, 1, 2, ..., N-1.$$

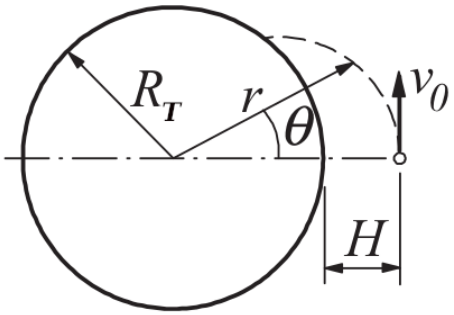
Calculer $z(t)$ avec la méthode *d'Euler implicite* pour $t \in [0,2]$ et pour un pas d'intégration $\Delta t = 0.1$ s.

e) Tracer dans un même graphique pour $t \in [0,2]$ et avec des pas d'intégration $\Delta t = 0.5, 0.1, 0.05, 0.01, 0.005$ s :

— La solution exacte : `sol_exacte(t, 0.05, 2)`
— $z(t)$ calculée par la méthode d'Euler explicite.
— $z(t)$ calculée par la méthode d'Euler explicite.

Que remarquez-vous pour les résultats trouvés ? Quelle est la méthode la plus proche de la solution exacte ?

6.6.3 Exercice 3 : Atterrissage d'un vaisseau spatial



Un vaisseau spatial est lancé à l'altitude $H = 772\ km$ au-dessus du niveau de la mer avec la vitesse $v_0 = 6700\ m/s$ dans la direction indiquée sur la figure ci-dessus. Les équations différentielles décrivant le mouvement du vaisseau spatial sont :

$$\ddot{r} = r\dot{\theta}^2 - \frac{GM_T}{r^2}$$
$$\ddot{\theta} = -\frac{2\dot{r}\dot{\theta}}{r}$$

où r et θ sont les coordonnées polaires du vaisseau spatial. Les constantes impliquées dans le mouvement sont :

— $G = 6.672 \times 10^{-11}\ m^3 kg^{-1} s^{-2}$ = constante gravitationnelle universelle.
— $M_T = 5.9742 \times 10^{24}\ kg$ = masse de la terre.
— $R_T = 6378.14\ km$ = rayon de la terre au niveau de la mer.

a) Dériver les équations différentielles du premier ordre et les conditions initiales de la forme $\dot{y} = F(t, y)$, $y(0) = b$.

b) Utiliser la méthode Runge-Kutta du quatrième ordre (RK4) pour intégrer les équations depuis le lancement jusqu'à ce que le vaisseau spatial touche la terre. Déterminez θ au site d'impact.

Chapitre 7

Résolution des équations aux dérivées partielles

7.1 Introduction

Il existe trois types d'équations aux dérivées partielles.

— Des équations telles que l'équation d'onde,

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2} \tag{7.1}$$

où $u(t,x)$ est une fonction de déplacement et c une vitesse constante, sont connues sous le nom d'équations hyperboliques.

— Des équations telles que l'équation de diffusion,

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(D\frac{\partial u}{\partial x}\right) \tag{7.2}$$

où $u(t,x)$ est le champ de densité et D le coefficient de diffusion sont appelés équations paraboliques. L'équation de Schrödinger en fonction du temps est un autre exemple d'équation parabolique.

— Des équations telles que l'équation de Poisson,

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = -\frac{\rho(x,y,z)}{\epsilon_0} \tag{7.3}$$

où $u(x,y,z)$ est une fonction potentielle et ρ/ϵ_0 est une source, appelées équations elliptiques. L'équation de Schrödinger indépendante du temps est un autre exemple d'équation elliptique.

On trouve des équations hyperboliques ou paraboliques dans les problèmes de valeurs initiales : la configuration du champ $u(t,x)$ est spécifiée à un moment initial et évolue dans le temps. Les équations elliptiques se retrouvent dans les problèmes de valeur limite : la valeur du champ $u(x,y,z)$ est spécifiée sur la limite d'une région et nous cherchons la solution à travers l'intérieur.

7.2 Équation de diffusion thermique

L'équation de diffusion thermique est historiquement liée à Joseph Fourier. Ce dernier naquit en 1768 à Auxerre, où il étudie dans une école militaire. Fin 1794, il est élève à Normale Sup et en 1795-1796, il enseigne la physique à Normale Sup et à l'X. Il débute ses travaux sur la chaleur en 1802 lorsqu'il est nommé préfet de l'Isère par Napoléon. Il publie sa théorie sur la chaleur (et l'analyse de Fourier) en 1822. Il est élu à l'Académie française en 1826 et décédé à Paris en 1830.

FIGURE 7.1 – Gravure du mathématicien Jean Baptiste Joseph Fourier (1768-1830). source 1 Article de **CNRS Le journal**, Joseph Fourier transforme toujours la science Article de CNRS

7.2.1 La conduction thermique dans les solides

Dans les solides, les molécules ou les atomes sont figés dans un réseau maillé qui empêchent les grands déplacements. Le seul mouvement possible est un mouvement de vibration autour de leur position d'équilibre dans le réseau. Le transfert thermique d'énergie se traduit par une augmentation plus ou moins grande de l'amplitude de ces vibrations. Lorsque l'énergie apportée est suffisante pour que l'amplitude de la vibration dépasse une certaine valeur dépendante du réseau, alors les molécules se libèrent du réseau et le solide fond et s'évapore...

Le cas des métaux est un peu particulier : eux possèdent des électrons de conduction, qui circulent librement sur le réseau du solide. Ces électrons se comportent comme les molécules d'un gaz (on parle de gaz d'électrons) et dans ce cas, à l'augmentation de l'amplitude des vibrations sur le réseau s'ajoute le transfert d'énergie cinétique des électrons rapides, "chauds" vers les électrons lents, "froids".

7.2.2 Notion de flux d'énergie

La notion de flux d'énergie est très courante en physique. Nous l'avons déjà rencontré sous le nom d'intensité du courant électrique. Dans ce cas, il s'agit du flux de charges électriques qui

traversent une surface unitaire donnée par unité de temps. Nous l'avons noté :

$$I_q = \frac{dQ}{dt} \tag{7.4}$$

Par analogie, on définit le flux thermique, plus exactement le flux d'énergie interne par $I_u = \frac{dU}{dt}$. C'est la quantité d'énergie interne U qui traverse, sans travail, une surface unitaire par unité de temps.

Pour l'exprimer plus précisément, on introduit un nouvel objet par analogie avec le courant électrique. Il s'agit du vecteur de courant volumique d'énergie interne sans travail noté classiquement $\vec{J_u}$ et souvent nommé improprement "vecteur courant thermique", ce qui nous permet d'écrire l'expression du flux de ce vecteur à travers une surface dS orientée vers l'extérieur par le vecteur normal $\vec{n}$, ce qui nous donne $I_u = \int_S \vec{J_u}.\vec{n}dS$. Le signe de I_u dépend du sens du flux à travers la surface. Il est négatif pour le flux entrant et positif pour le flux sortant (pensez au signe du produit scalaire sous l'intégrale...).

7.2.3 L'expression de la loi de Fourier

Après ces préambules utiles, venons en à la loi de Fourier proprement dite. La conduction thermique est un transfert thermique spontané d'une région de température élevée vers une région de température plus basse, et est décrite par la loi dite de Fourier établie mathématiquement par Jean-Baptiste Biot en 1804 puis expérimentalement par Fourier en 1822 : la densité de flux de chaleur est proportionnelle au gradient de température.

Pour faire simple, ici et dans la suite, on va se placer dans le cadre d'un problème unidimensionnel, c'est à dire que le transfert thermique d'énergie se fait sur une dimension Ox.

La loi de Fourier relie, après constat expérimental, le vecteur de courant volumique d'énergie interne sans travail $\vec{J_u}$ avec le gradient de température $\vec{grad}T$. La relation est linéaire et s'écrit $\vec{J_u} = -\kappa\ \vec{grad}T$. La linéarité de l'équation n'est due qu'aux approximations que nous avons fixé à son domaine de validité, comme pour la loi d'Ohm.

Le paramètre κ est appelé conductivité thermique, toujours positif, de dimension $W.m^{-1}.K^{-1}$. Notez la présence du signe $-$, qui résulte du second principe de la thermodynamique : le flux d'énergie va des régions aux températures les plus hautes vers les régions aux températures les plus basses.

Dans notre hypothèse d'un problème unidimensionnel, en développant le gradient, on obtient l'équation $\vec{J_u} = -\kappa\frac{\partial T}{\partial x}\vec{u}$, soit en projetant sur Ox, $J_u(x,t) = -\kappa\frac{\partial T(x,t)}{\partial x}$ Nous obtenons une équation à deux variables, la position x et le temps t, avec une dérivée partielle.

7.2.4 Comment obtenir cette équation ?

Nous allons l'établir en utilisant la loi de Fourier décrite ci-dessus et le principe de conservation d'énergie. Nous resterons dans le cadre du problème unidimensionnel, sachant que l'extension en dimension 2 ou 3 n'est pas très compliquée, mais trop lourde pour notre étude.

Considérons un élément de volume dV orienté selon l'axe Ox de propagation du flux d'énergie, limité par deux surfaces dS, l'une entrante et l'autre sortante, et d'épaisseur dx. Appelons $J_u\vec{E}$ le vecteur de courant d'énergie volumique entrant et $J_u\vec{S}$ le vecteur de courant d'énergie volumique sortant. Supposons que ces deux vecteurs soient normaux aux surfaces entrantes et sortantes.

Cet élément de volume dV est immobile et son énergie potentielle n'est pas modifiée par hypothèse. Selon le premier principe de la thermodynamique, la variation d'énergie interne dU n'est donc attribuable qu'à la variation dQ, le transfert thermique d'énergie.

Calculons la variation d'énergie interne dans ce volume dV, de masse volumique ρ, en utilisant la définition de $\vec{J_u}$ donnée plus haut dans le cas unidimensionnel. Nous obtenons après simplification, en égalant dU et dQ :

$$\frac{\partial(\rho u)}{\partial t}dx = -J_{u,x}(x+dx,t) + J_{u,x}(x,t) \tag{7.5}$$

En remarquant que $J_{u,x}(x+dx,t) - J_{u,x}(x,t) = \frac{\partial J_{u,x}}{\partial x}dx$, on obtient finalement :

$$\frac{\partial(\rho u)}{\partial t} = -\frac{\partial J_{u,x}}{\partial x} \tag{7.6}$$

Dans cette dernière équation, remplaçons dans le terme de droite $J_{u,x}$ par sa définition donnée par la loi de Fourier, on obtient :
$\frac{\partial(\rho u)}{\partial t} = -\frac{\partial}{\partial x}(-\kappa\frac{\partial T}{\partial x})$ ou en condensant l'écriture :

$$\frac{\partial(\rho u)}{\partial t} = \kappa\frac{\partial^2 T}{\partial x^2} \tag{7.7}$$

Vous aurez noté ici, si vous êtes attentifs, que j'ai considéré que κ était constant puisque je l'ai sorti de la dérivée sans autre forme de procès ! C'est un peu osé, et vrai seulement si le milieu est isotrope (le matériaux est homogène) et si l'on ne chauffe pas trop fort ou trop vite, parce que sinon, il devient dépendant de la température.

Reste maintenant à traiter le terme de gauche. Pour ce faire, je vais faire appel à l'expression de la capacité calorique qui relie les variations de l'énergie avec les variations de température. On peut donc écrire que, si ρu désigne l'énergie interne volumique :

$$\frac{\partial(\rho u)}{\partial t} = \rho c_v\frac{\partial T}{\partial t} \tag{7.8}$$

avec c_v la capacité thermique massique à volume constant.

En reportant cette expression dans le terme de gauche de notre équation, j'obtiens :

$$\rho c_v\frac{\partial T}{\partial t} = \kappa\frac{\partial^2 T}{\partial x^2} \tag{7.9}$$

soit en regroupant les termes constants à droite de l'équation :

$$\frac{\partial T}{\partial t} = \frac{\kappa}{\rho c_v}\frac{\partial^2 T}{\partial x^2} \tag{7.10}$$

Pour simplifier l'écriture, je vais appeler D le rapport $\frac{\kappa}{\rho c_v}$. Ce paramètre est la diffusivité thermique du matériau constituant notre élément de volume. La dimension de D est $m^2.s^{-1}$. Finalement, j'obtiens l'équation :

$$\frac{\partial T}{\partial t} = D\frac{\partial^2 T}{\partial x^2} \tag{7.11}$$

qui constitue l'équation de diffusion thermique !

7.2.5 Résolution numérique de l'équation de diffusion thermique

Le modèle physique. Nous allons considérer une barre métallique homogène de faible diamètre, de telle sorte que nous puissions négliger ses dimensions spatiales autre que sa longueur. Autrement dit, je m'arrange pour avoir un modèle approximativement 1D.

Nous allons considérer que cette barre solide de longueur $L = 1m$ de coefficient de diffusion thermique $D \approx 0.5\ m^2.s^{-1}$. La barre est initialement préparée dans un état de température $T(x, t < 0) = T_0(x) = 100C^\circ$.

À l'instant $t \geq 0$, les extrémités de la barre spnt misent en contact avec deux sources de températures identiques $T_{extr} = 0c^\circ$, donc nous aurons :

$$T(x = 0, t \geq 0) = T(x = L, t \geq 0) = T_{extr}$$

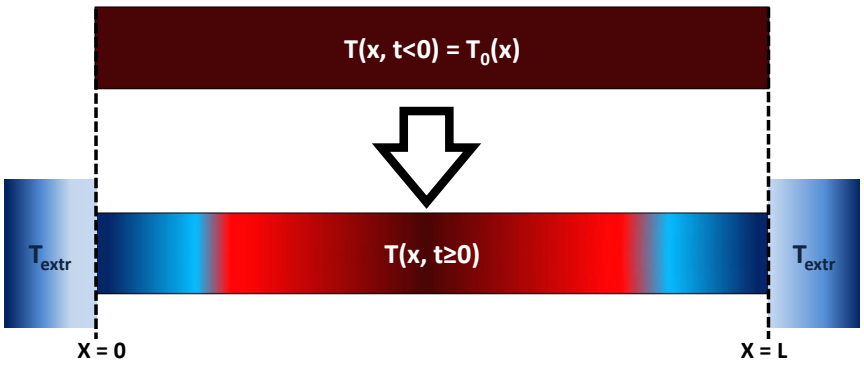


FIGURE 7.2 – Choc thermique sur une barre de longueur L.

La température $T(x, t)$ de la barre est solution de l'équation (7.11) de la diffusion thermique à 1D

Principe de la résolution numérique. On recherche une solution numérique à ce problème par la classique méthode des *différences finies*.

Supposons que nous cherchions l'évolution de $T(x, t)$ sur une durée totale $\tau = 1s$:

— La barre est spatialement discrétisée en N_x tronçons de longueur égale $\Delta x = \frac{L}{N_x}$. Ainsi, l'abscisse discrète x_m est : $x_m = m * \Delta x$ avec $m \in [0, N_x]$.

— De même, la durée totale de l'évolution est discrétisée en N_t intervalles de durée $\Delta t = \frac{\tau}{N_t}$. Ainsi, l'instant "discret" t_n est : $t_n = n * \Delta t$ avec $n \in [0, N_t]$.

On peut, par conséquent, poser la température discrétisée :

$$T_{m,n} = T(m * \Delta x, n * \Delta t)$$

Discrétisation du terme $\frac{\partial T}{\partial t}$. Pour x fixe ($x = x_m$), $T(x_m, t)$ est de classe C^1 sur $[0, \tau]$ par rapport au temps.

Soit le développement de Taylor à l'ordre 1 :

$$T(x_m, t_{n+1}) = T(x_m, t_n + \Delta t) \approx T(x_m, t_n) + \Delta t \frac{\partial T(x_m, t_n)}{\partial t} + \mathcal{O}(\Delta t^2)$$

Nous revenons à l'expression explicite d'Euler déjà abordée dans le chapitre précédent. On peut donc avoir l'expression discrétisée du terme $\frac{\partial T}{\partial t}$:

$$\frac{\partial T(x_m, t_n)}{\partial t} = \frac{T(x_m, t_{n+1}) - T(x_m, t_n)}{\Delta t} \tag{7.12}$$

Discrétisation du terme $\frac{\partial^2 T}{\partial x^2}$. Pour t fixe ($t = t_n$), $T(x, t_n)$ est de classe C^2 sur $[0, L]$ par rapport à x.

Soit le développement de Taylor à l'ordre 2 :

$$T(x_{m+1}, t_n) = T(x_m + \Delta x, t_n) \approx T(x_m, t_n) + \Delta x \frac{\partial T(x_m, t_n)}{\partial x} + \frac{\Delta x^2}{2!} \frac{\partial^2 T(x_m, t_n)}{\partial x^2} + \mathcal{O}(\Delta t^3)$$

La dérivée première $\frac{\partial T(x_m, t_n)}{\partial x}$ ne figure pas dans l'équation initiale (Eq. (7.11)), il faut donc l'éliminer !

> **Note:** L'idée est de faire une addition des développements en $(x_m + \Delta x)$ et en $(x_m - \Delta x)$.

$$T(x_{m-1}, t_n) = T(x_m - \Delta x, t_n) \approx T(x_m, t_n) - \Delta x \frac{\partial T(x_m, t_n)}{\partial x} + \frac{\Delta x^2}{2!} \frac{\partial^2 T(x_m, t_n)}{\partial x^2} + \mathcal{O}(\Delta t^3)$$

$$T(x_{m+1}, t_n) + T(x_{m-1}, t_n) \approx 2T(x_m, t_n) + \frac{2\Delta x^2}{2!} \frac{\partial^2 T(x_m, t_n)}{\partial x^2} + \mathcal{O}(\Delta t^3)$$

D'où l'expression discrétisée du terme $\frac{\partial^2 T}{\partial x^2}$:

$$\frac{\partial^2 T(x_m, t_n)}{\partial x^2} = \frac{T(x_{m+1}, t_n) - 2T(x_m, t_n) + T(x_{m-1}, t_n)}{\Delta x^2} \tag{7.13}$$

Remplaçons les équations (7.12) et (7.13) dans l'équation (7.11) de la diffusion thermique. Ainsi l'équation de la diffusion thermique 1D discrétisée s'écrit :

$$\frac{T(x_m,t_{n+1})-T(x_m,t_n)}{\Delta t} \approx D\frac{T(x_{m+1},t_n)-2T(x_m,t_n)+T(x_{m-1},t_n)}{\Delta x^2} \tag{7.14}$$

D'où l'on tire finalement la relation de récurrence permettant d'obtenir la température en x_m à l'instant t_{n+1} en fonction des températures $T_{m,n}$, $T_{m+1,n}$ et $T_{m-1,n}$ calculées à l'instant t_n :

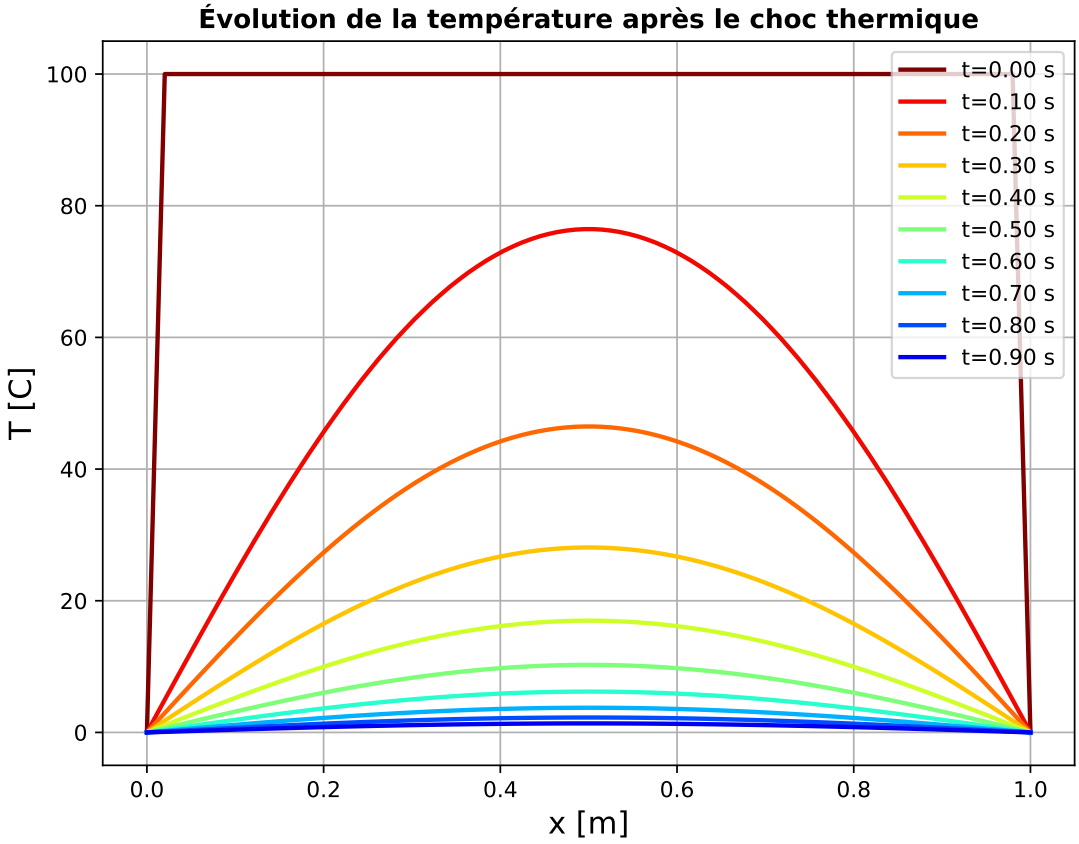
$$T(x_m,t_{n+1}) \approx T(x_m,t_n)+D\frac{\Delta t}{\Delta x^2}\left[T(x_{m+1},t_n)-2T(x_m,t_n)+T(x_{m-1},t_n)\right] \tag{7.15}$$

> **Note:** On peut écrire l'expression (7.15) avec la notation suivante :
>
> $$T_{m,n+1} \approx T_{m,n}+D\frac{\Delta t}{\Delta x^2}\left[T_{m+1,n}-2T_{m,n}+T_{m-1,n}\right]$$

Code Python. Le programme Python pour ce problème est :

```
## NOM DU PROGRAMME: EDP_DiffChal.py
#% IMPORTATION
import numpy as np
import matplotlib.pyplot as plt
# DONNÉES NUMÉRIQUES
L = 1 # Longueur de la barre [m]
Nx = 100 # Nombre de tronçons
tau = 1 # Durée totale de l'évolution [s]
Nt = 10000 # Nombre d'intervalles de temps
dx = L/Nx # Longueur du tronçon
dt = tau/Nt # Intervalle élémentaire de temps
D = 0.5 # Coefficient de diffusion thermique
Tb = 100 # Température initiale de la barre
Textr = 0 # Température des extrémités
# Construction de axe des abscisses (variable espace x)
x = np.linspace(0, L, Nx)
# Construction CI ,CL (2 exemples pour la CI)
T = [Textr] + (Nx - 2)*[Tb] + [Textr] # Echelon de temperature
#T=Textr + (Tb-Textr ) * np.sin(np.pi*x/L) # Arche sinusoidale temperature
# Construction du tableau vierge des accroissements de temperature
accroissT = np.zeros(Nx)
T[0] = 0
plt.figure(figsize=(8, 6)) # Créer le graphique
# Corps de la résolution
for n in range(Nt): # Boucle d'évolution de temps pas à pas
    for m in range(1, Nx-1): #boucle de calcul de accroissement de temperature pour chaque abscisse
        accroissT[m] = ((dt*D)/(dx**2))*(T[m-1]-2*T[m]+T[m+1])
    for m in range(1, Nx-1): # Boucle de calcul de T instant  suivant
        T[m] += accroissT[m]
    #Trace tous les 1000 intervalles de temps
    if (n%1000 == 0):
        plotlabel ="t=%1.2f s"%(n*dt)
        plt.plot(x, T, label=plotlabel,lw = 4, color = plt.get_cmap('jet')(1-n/Nt))

plt.grid()
plt.xlabel("x [m]", fontsize=14)
plt.ylabel("T [C]", fontsize=14)
plt.title("Évolution de la température après le choc thermique", weight ="bold")
plt.legend(loc=1)
plt.savefig("chaleur.png"); plt.savefig("chaleur.pdf")
plt.tight_layout()
plt.show()
```

Lors de l'exécution de ce code, nous aurons la figure ci-dessous :



7.3 L'équation de Laplace

Nous considérons maintenant les équations elliptiques. L'équation de Poisson en trois dimensions est :

$$\frac{\partial^2 u(x,y,z)}{\partial x^2} + \frac{\partial^2 u(x,y,z)}{\partial y^2} + \frac{\partial^2 u(x,y,z)}{\partial z^2} = -\frac{\rho(x,y,z)}{\epsilon_0} \tag{7.16}$$

où $u(x,y,z)$ est le champ de potentiel électrique et $\rho(x,y,z)$ est la densité de charge. Par souci de simplicité, cependant, nous étudierons l'équation de Laplace en deux dimensions :

$$\frac{\partial^2 u(x,y)}{\partial x^2} + \frac{\partial^2 u(x,y)}{\partial y^2} = 0 \tag{7.17}$$

sur le carré $0 \leq x \leq L$ et $0 \leq y \leq L$ avec une paroi du carré maintenue (la paroi à $y = L$) à un potentiel de $V_0 = 1V$ et les autres parois mises à la terre à $0V$.

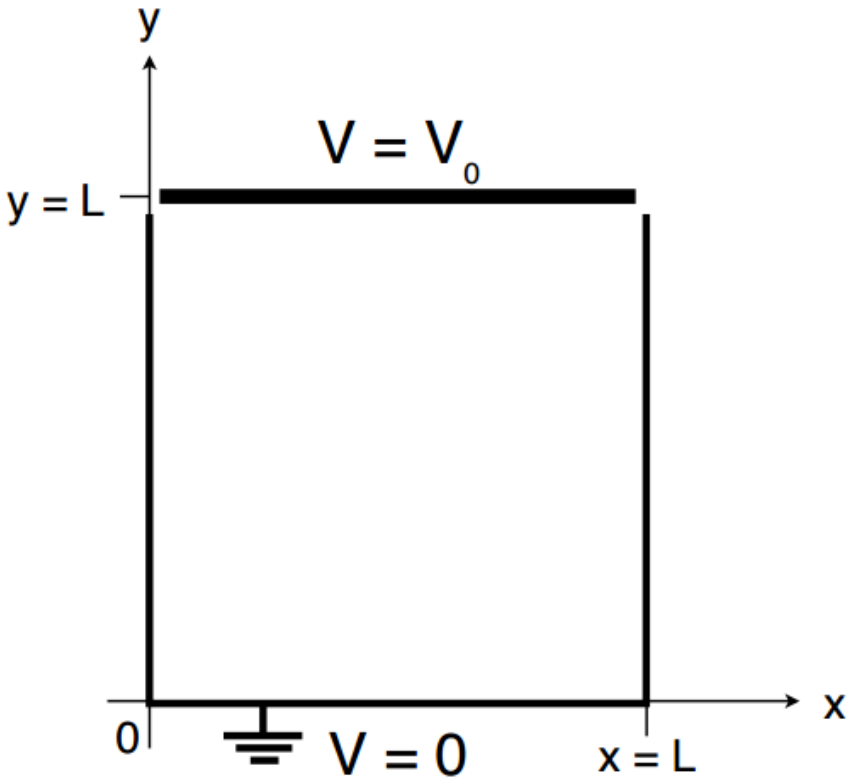


FIGURE 7.3 – Exemple de problème de valeur limite pour l'équation de Laplace en deux dimensions.

Nous utilisons la séparation des variables

$$u(x,y) = X(x)Y(y) \tag{7.18}$$

pour exprimer l'équation de Laplace comme les équations différentielles ordinaires

$$-\frac{X"}{X} = \frac{Y"}{Y} = k^2 \tag{7.19}$$

où k est une constante de séparation. Les conditions aux limites sont $X(0) = 0$, $X(L) = 0$, $Y(0) = 0$ et $Y(L) = V_0$. Les solutions pour X avec ces conditions aux limites sont

$$X(x) \propto sin(\frac{n\pi x}{L}) \quad pour\ n = 1, 2, 3, \ldots \tag{7.20}$$

et les constantes de séparation autorisées sont $k_n = n\pi/L$. Les solutions pour Y seront une combinaison linéaire des fonctions sinus hyperbolique et cosinus hyperbolique, mais comme seule la fonction sinus hyperbolique disparaît à $x = 0$, notre solution est de la forme

$$u(x,y) = \sum_{n=1}^{\infty} c_n sin(\frac{n\pi x}{L}) sinh(\frac{n\pi y}{L}) \tag{7.21}$$

La condition aux limites finale, $u(x,L) = V_0$, détermine alors les coefficients c_n : on a

$$\sum_{n=1}^{\infty} c_n sinh(n\pi) sin(\frac{n\pi x}{L}) = V_0 \tag{7.22}$$

Nous multiplions les deux côtés par $sin(m\pi x = L)$ et intégrons de $x = 0$ à $x = L$ pour obtenir

$$\frac{L}{2} sinh(m\pi) = \begin{cases} \frac{2LV_0}{m\pi} & \text{pour m impair} \\ 0 & \text{autrement.} \end{cases} \tag{7.23}$$

Nous avons donc

$$c_m = \begin{cases} \frac{4V_0}{m\pi \ sinh(m\pi)} & \text{pour m impair} \\ 0 & \text{autrement.} \end{cases} \tag{7.24}$$

et la solution de l'équation de Laplace est

$$u(x,y) = 4V_0 \sum_{\substack{n=1 \\ n \ impair}}^{\infty} \frac{sin(n\pi x/L)}{n\pi} \frac{sinh(n\pi y/L)}{sinh(n\pi)} \tag{7.25}$$

Un grand nombre de termes dans cette série sont nécessaires pour calculer avec précision le champ près du mur près de $y = L$ (et surtout aux coins).

La méthode de relaxation peut être utilisée pour les équations elliptiques de la forme

$$\hat{L}u = \rho \tag{7.26}$$

où $\hat{L}$ est un opérateur elliptique et ρ est un terme source. L'approche consiste à prendre une distribution initiale u qui ne résout pas nécessairement l'équation elliptique et à lui permettre de se détendre à la solution de l'équation en faisant évoluer l'équation de diffusion

$$\frac{\partial u}{\partial t} = \hat{L}u - \rho \tag{7.27}$$

Aux temps tardifs, $t \to 0$, la solution s'approchera asymptotiquement de la solution stationnaire de l'équation elliptique. Pour le problème en question ($\rho = 0$ et $\hat{L}$ est l'opérateur laplacien bidimensionnel), la méthode FTCS appliquée à l'équation de diffusion conduit à

$$u_{j,k}^{n+1} = u_{j,k}^n + \left[\frac{u_{j+1,k}^n - 2u_{j,k}^n + u_{j-1,k}^n}{(\Delta x)^2} + \frac{u_{j,k+1}^n - 2u_{j,k}^n + u_{j,k-1}^n}{(\Delta y)^2} \right] \Delta t \tag{7.28}$$

où $u_{j,k}^n = u^n(j\Delta x, k\Delta y)$ et n indique l'itération. Pour simplifier, nous prenons $\Delta x = \Delta y = \Delta$ donc nous avons

$$u_{j,k}^{n+1} = (1-\omega)u_{j,k}^n + \frac{\omega}{4}(u_{j+1,k}^n + u_{j-1,k}^n + u_{j,k+1}^n + u_{j,k-1}^n) \tag{7.29}$$

où $\omega = 4\Delta t/\Delta^2$. La stabilité de l'équation de diffusion limite l'ampleur des ω. Nous pouvons déterminer la valeur maximale de ω en utilisant une analyse de stabilité de von Neumann, mais maintenant nous nous limiterons aux modes propres spatiaux qui satisfont aux conditions aux limites de Dirichlet pour la partie homogène de la solution. Notre approche est donc

$$u_{j,k}^n = u^0 \zeta^n(m_x, m_y) sin(\frac{m_x \pi j \Delta}{L}) sin(\frac{m_y \pi k \Delta}{L}) \tag{7.30}$$

où m_x et m_y sont respectivement les numéros de mode dans les directions x et y. En substituant cette approche à l'Eq. (7.29) on trouve

$$\zeta(m_x, m_y) = 1 - \omega + \frac{\omega}{2}\left(cos(\frac{m_x \pi \Delta}{L}) + cos(\frac{m_y \pi \Delta}{L})\right) \tag{7.31}$$

et nous voyons que la stabilité est atteinte, c'est-à-dire que $|\zeta(m_x, m_y)| \leq 1$ pour tout mode donné par (m_x, m_y), si $\omega \leq 1$. Si nous prenons maintenant la plus grande valeur de $\Delta t = \Delta^2/4$ permise pour une itération stable correspondant à $\omega = 1$, nous obtenons l'itération suivante schème :

$$u^{n+1}_{j,k} = \frac{1}{4}(u^n_{j+1,k} + u^n_{j-1,k} + u^n_{j,k+1} + u^n_{j,k-1}) \tag{7.32}$$

On voit ici que la valeur du champ à un point de réseau donné (j,k) à l'étape $n+1$ est égale à la moyenne des valeurs du champ aux points voisins à l'étape n. Ceci est connu comme *la méthode de Jacobi*.

Le programme `Laplace_relax.py` implémente la méthode de Jacobi pour notre problème de modèle. Le nombre de points de grille de chaque côté, N, est entré. Les résultats obtenus pour $N = 20$ points sont présentés sur la figure 7.4.

```
## NOM DU PROGRAMME: Laplace_relax.py
#% IMPORTATION
import numpy as np
import matplotlib.pyplot as plt
import mpl_toolkits.mplot3d
eps = 1e-5 # erreur fractionnaire autorisée
L = 1.0 # longueur de chaque côté
N = int(input('nombre de points de grille sur un côté -> '))
dy = dx = L/(N-1.0)
x = np.array(range(N))*dx
y = np.array(range(N))*dy
(x, y) = np.meshgrid(x, y)
u0 = np.zeros((N, N))
u1 = np.zeros((N, N))
# conditions aux limites
for j in range(N):
    u1[j,N-1] = u0[j,N-1] = 1.0

# préparer l'animation
image = plt.imshow(u0.T, origin='lower', extent=(0.0, L, 0.0, L))
n = 0 # nombre d'itérations
err = 1.0 # erreur moyenne par site
while err > eps:
    # mettre à jour le tracé animé
    image.set_data(u0.T)
    plt.title('itération %d'%n)
    plt.tight_layout()
    plt.show()
    plt.pause(0.001)
    # prochaine itération en raffinement
    n = n+1
    err = 0.0
    for j in range(1, N-1):
        for k in range(1, N-1):
            u1[j,k] = (u0[j-1,k]+u0[j+1,k]+u0[j,k-1]+u0[j,k+1])/4.0
            err += abs(u1[j,k]-u0[j,k])
    err /= N**2
    # permutez les anciens et les nouveaux tableaux pour la prochaine itération
    (u0, u1) = (u1, u0)

# tracé de surface de la solution finale

fig = plt.figure()
axis = fig.gca(projection='3d', azim=-60, elev=20)
```

```
surf = axis.plot_surface(x, y, u0.T, rstride=1, cstride=1, cmap='viridis')
wire = axis.plot_wireframe(x, y, u0.T, rstride=1+N//50, cstride=1+N//50,
                           color = "r", linewidth=0.5, alpha = 0.5)
axis.contour(x, y, u0.T, 10, zdir='z', offset=-1.0)
axis.set_xlabel('x')
axis.set_ylabel('y')
axis.set_zlabel('u')
axis.set_zlim(-1.0, 1.0)
fig.colorbar(surf)
plt.tight_layout()
plt.savefig("Laplace_relax.png"); plt.savefig("Laplace_relax.pdf")
plt.show()
```

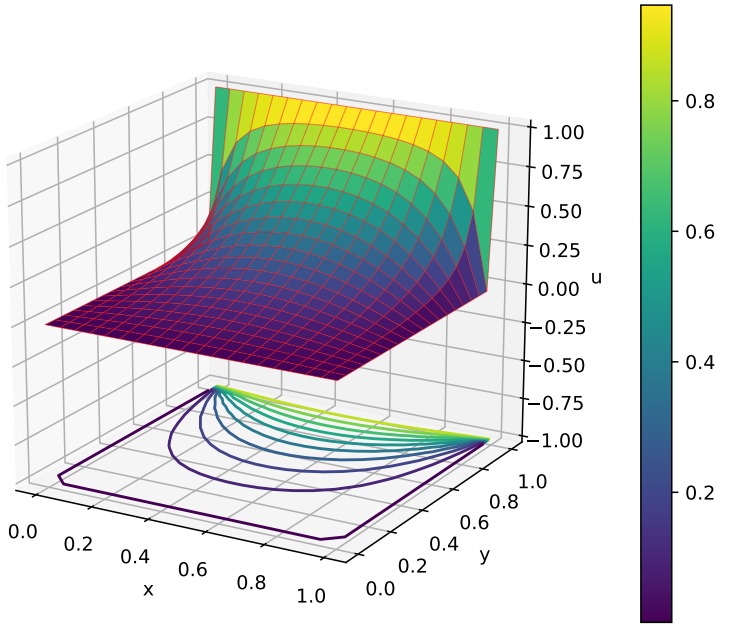


FIGURE 7.4 – Résultats de l'exécution du programme `Laplace\_relax.py` avec les paramètres N = 20 points de grille de chaque côté. La solution est obtenue après 386 itérations.

Dans le programme `Laplace_relax.py`, l'itération s'est poursuivie jusqu'à ce que l'erreur moyenne par point de maillage soit inférieure à $\epsilon = 10^{-5}$ où l'erreur a été estimée en prenant la différence entre l'ancienne valeur à l'étape n et la nouvelle valeur à l'étape $n+1$. Le nombre d'itérations nécessaires à la convergence dépend du mode propre en décomposition le plus lent de l'itération. Le module du mode de décomposition le plus lent est connu sous le nom de *rayon spectral*

$$\rho = max_{m_x,m_y}|\zeta(m_x, m_y)|. \tag{7.33}$$

De l'Eq. (7.31) nous voyons que pour la méthode de Jacobi avec $\omega = 1$ nous avons

$$\rho = \rho_J = cos(\frac{\pi\Delta}{L}). \tag{7.34}$$

ce qui correspond au mode $m_x = m_y = 1$. Chaque itération multiplie l'erreur résiduelle dans ce mode le moins amorti par un facteur de module ρ et donc le nombre d'itérations nécessaires pour atteindre la tolérance d'erreur souhaitée ϵ sera

$$n = \frac{ln\epsilon}{ln\rho}. \tag{7.35}$$

Pour la méthode Jacobi, $\rho_J = 1 - \frac{1}{2}(\pi/N)$ et ainsi

$$n \approx \frac{2|ln\epsilon|}{\pi^2} N^2. \tag{7.36}$$

Notez que si nous doublons le nombre de points de grille N, nous avons besoin de quatre fois plus d'itérations pour converger. Pour les problèmes pratiques, la méthode de Jacobi converge trop lentement pour être utile.

Pour progresser, considérons à nouveau l'Eq. (7.29) et notons que si nous imaginons que notre grille de calcul est divisée en points clairs et sombres décalés, comme le montre la figure 7.5, puis pour mettre à jour la valeur d'un point blanc, nous n'avons besoin que de la valeur actuelle à ce point blanc et des valeurs des points sombres voisins et vice versa pour mettre à jour la valeur d'un point sombre.

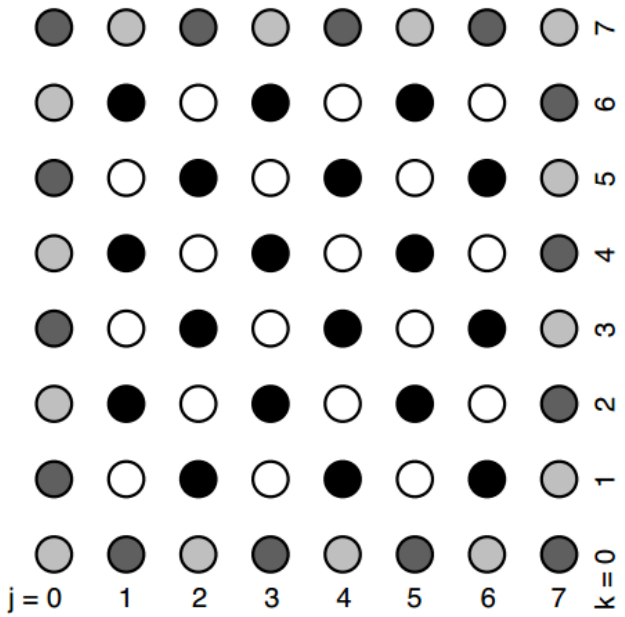


FIGURE 7.5 – Un réseau décalé de points sombres et clairs pour une utilisation en sur-relaxation successive. Les points gris font partie de la frontière et ne sont pas évolués. Pour mettre à jour un point blanc ne nécessite que la valeur précédente du point blanc et des points sombres environnants et de même pour mettre à jour un point noir ne nécessite que la valeur précédente des points noirs et des points clairs environnants.

Ainsi, nous pouvons adopter une approche échelonnée où nous mettons à jour tous les points blancs, puis nous mettons à jour tous les points noirs et les deux étapes peuvent être effectuées sur place. Pour les étapes où n est un entier, nous calculons les valeurs des points blancs en utilisant la formule

$$u_{j,k}^{n+1} = (1-\omega)u_{j,k}^{n} + \frac{\omega}{4}(u_{j+1,k}^{n+1/2} + u_{j-1,k}^{n+1/2} + u_{j,k+1}^{n+1/2} + u_{j,k-1}^{n+1/2}) \tag{7.37}$$

puis pour les étapes où n est un demi-entier, nous utilisons la même formule pour calculer les valeurs des points noirs. Nous pouvons répéter l'analyse de stabilité en utilisant l'approche de

l'équation. (7.30) et nous trouvons

$$\zeta^{1/2}(m_x, m_y) = \frac{\omega c \pm \sqrt{\omega^2 c^2 - 4(\omega - 1)}}{2} \tag{7.38}$$

où

$$c = \frac{1}{2}(cos(\frac{m_x \pi \Delta}{L}) + cos(\frac{m_y \pi \Delta}{L})) \tag{7.39}$$

Cela révèle que le schéma de l'Eq. (7.37) est stable pour $0 < \omega < 2$. Lorsque $\omega = 1$, la méthode est connue sous le nom de méthode Gauss-Seidel, qui converge un peu plus rapidement que la méthode Jacobi. Pour $\omega > 1$, nous avons accéléré la convergence (par rapport à la relaxation) qui est connue sous le nom de sur-relaxation successive ou SOR (Successive Over-Relaxation, en anglais). Le paramètre ω est connu comme le paramètre de sur-relaxation.

Il existe une valeur optimale pour le paramètre de sur-relaxation pour lequel le rayon spectral est minimisé. Si nous nous concentrons sur le mode le moins amorti pour lequel $m_x = m_y = 1$, nous avons $c = \rho_J$ et donc le rayon spectral en fonction de ω peut être écrit comme

$$\rho(\omega) = \begin{cases} \left[\frac{1}{2}\omega\rho_J + \frac{1}{2}\sqrt{\omega^2\rho_J^2 - 4(\omega - 1)}\right]^2 & pour\ 0 < \omega \leq \omega_{opt} \\ \omega - 1 & pour\ \ \omega_{opt} \leq \omega < 2 \end{cases} \tag{7.40}$$

où ω_{opt} est le choix optimal qui minimise ρ,

$$\omega_{opt} = \frac{2}{1 + \sqrt{1 - \rho_J^2}} = \frac{2}{1 + sin(\pi\Delta/L)} \quad pour\ \rho_J = cos(\pi\Delta/L). \tag{7.41}$$

Ainsi, ou le choix optimal du paramètre de sur-relaxation, $\omega = \omega_{opt} \approx \pi = N$, le rayon spectral est

$$\rho(\omega_{opt}) = \rho_{opt} = \frac{1 - sin(\pi\Delta/L)}{1 + sin(\pi\Delta/L)} \approx 1 - \frac{2\pi}{N} \tag{7.42}$$

et le nombre d'itérations nécessaires pour réduire l'erreur à une certaine tolérance ϵ est

$$n \approx \frac{ln\epsilon}{ln\rho_{opt}} \approx \frac{|ln\epsilon|}{2\pi} N. \tag{7.43}$$

Maintenant, le nombre d'itérations est proportionnel à N plutôt qu'à N^2, donc la convergence est atteinte beaucoup plus rapidement pour les grandes valeurs de N.

Le programme `Laplace_surrelax.py` est une modification de `Laplace_relax.py` qui implémente une sur-relaxation successive. Le programme diffère dans de nombreux endroits, il est donc répertorié dans son intégralité. Les résultats pour $N = 100$ points le long d'un côté sont affichés sur la figure 7.6. La convergence se produit en 137 itérations.

```
## NOM DU PROGRAMME: Laplace_surrelax.py
#% IMPORTATION
import numpy as np
import matplotlib.pyplot as plt
import mpl_toolkits.mplot3d
eps = 1e-5 # erreur fractionnaire autorisée
L = 1.0 # longueur de chaque côté
N = int(input('nombre de points de grille sur un côté -> '))
```

```
dy = dx = L/(N-1.0)
x = np.array(range(N))*dx
y = np.array(range(N))*dy
(x, y) = np.meshgrid(x, y)
u = np.zeros((N, N))
# conditions aux limites
for j in range(N):
    u[j,N-1] = 1.0
# calculer le paramètre de sur-relaxation
omega = 2.0/(1.0+np.sin(np.pi*dx/L))
# pixels blancs et noirs: les blancs ont j+k pairs; les noirs ont j+k impairs
blanc = [(j, k) for j in range(1, N-1) for k in range(1, N-1) if (j+k)%2 == 0]
noir = [(j, k) for j in range(1, N-1) for k in range(1, N-1) if (j+k)%2 == 1]
# préparer l'animation
image = plt.imshow(u.T, origin='lower', extent=(0.0, L, 0.0, L))
n = 0 # nombre d'itérations
err = 1.0 # erreur moyenne par site
while err > eps:
    # mettre à jour le tracé animé
    image.set_data(u.T)
    plt.title('itération %d'%n)
    plt.tight_layout()
    plt.show()
    plt.pause(0.001)
    # prochaine itération en raffinement
    n = n+1
    err = 0.0
    for (j, k) in blanc+noir: # boucle sur pixels blancs puis pixels noirs
        du = (u[j-1,k]+u[j+1,k]+u[j,k-1]+u[j,k+1])/4.0-u[j,k]
        u[j,k] += omega*du
        err += abs(du)
    err /= N**2
# tracé de surface de la solution finale
fig = plt.figure()
axis = fig.gca(projection='3d', azim=-60, elev=20)
surf = axis.plot_surface(x, y, u.T, rstride=1, cstride=1, cmap='viridis')
wire = axis.plot_wireframe(x, y, u.T, rstride=1+N//50, cstride=1+N//50,
                           color = "r", linewidth=0.5, alpha = 0.5)
axis.contour(x, y, u.T, 10, zdir='z', offset=-1.0)
axis.set_xlabel('x')
axis.set_ylabel('y')
axis.set_zlabel('u')
axis.set_zlim(-1.0, 1.0)
fig.colorbar(surf)
plt.tight_layout()
plt.savefig("Laplace_surrelax.png"); plt.savefig("Laplace_surrelax.pdf")
plt.show()
```

Comme dernier exemple, résolvons le potentiel électrique produit par une charge ponctuelle centrée dans un cube avec des bords de longueur $2L$ dans laquelle les faces du cube sont mises à la terre comme le montre la figure 7.7.

Il s'agit maintenant d'un problème tridimensionnel que nous pouvons à nouveau résoudre en utilisant une sur-relaxation successive, et notre équation d'itération comprend désormais également un terme source :

$$u_{i,j,k}^{n+1} = (1-\omega)u_{i,j,k}^{n} + \frac{\omega}{6}(u_{i+1,j,k}^{n+1/2} + u_{i-1,j,k}^{n+1/2} + u_{i,j+1,k}^{n+1/2} + u_{i,j-1,k}^{n+1/2} + u_{i,j,k+1}^{n+1/2} + u_{i,j,k-1}^{n+1/2}) + \frac{\omega}{6}\frac{\rho_{i,j,k}}{\epsilon_0} \tag{7.44}$$

Encore une fois, nous divisons le réseau de points de la grille en pixels «blancs» et «noirs» alternés et résolvons les pixels blancs sur les pas entiers et les pixels noirs sur les pas demi-entiers. La charge ponctuelle nous donne une densité de charge que nous considérons comme étant

$$\rho_{i,j,k} = \begin{cases} \dfrac{q}{\Delta^3} & pour\ i=j=k=(N-1)/2 \\ 0 & \text{autrement} \end{cases} \tag{7.45}$$

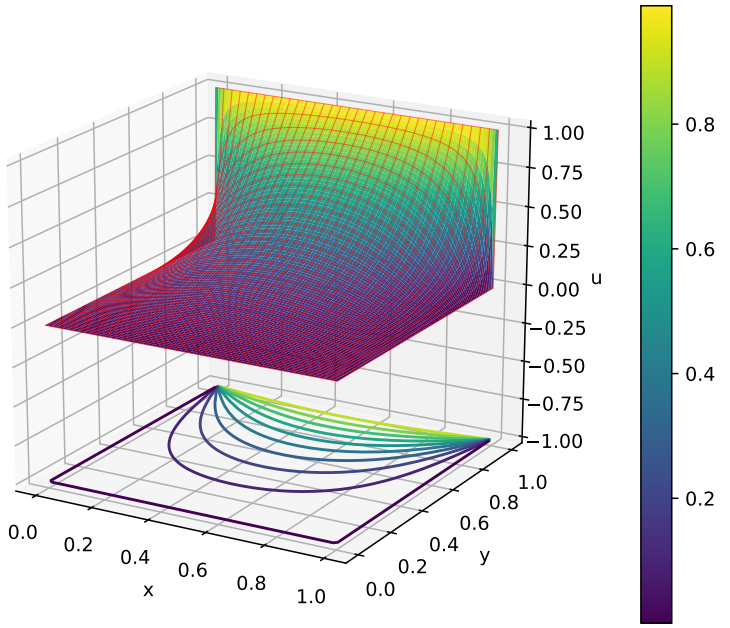


FIGURE 7.6 – Résultats de l'exécution du programme `LaplaceJacobi\_surrelax.py` avec les paramètres N = 100 points de grille le long de chaque côté. La solution est obtenue après 137 itérations.

et nous devons être sûrs de choisir une valeur impaire pour N afin qu'il y ait un point de grille au centre exact de la boîte.

Le programme `charge.py` répertorié ci-dessous calcule le champ de potentiel électrique dans la boîte mise à la terre pour une unité $q = \epsilon_0 = 1$ charge. Les résultats sont présentés sur la figure 7.8.

```
## NOM DU PROGRAMME: charge.py
#% IMPORTATION
import numpy as np
import matplotlib.pyplot as plt
import mpl_toolkits.mplot3d
eps = 1e-5 # erreur fractionnaire autorisée
L = 1.0 # longueur de chaque côté
N = int(input('nombre de points de grille sur un côté -> '))
dz = dy = dx = 2.0*L/(N-1.0)
x = -L + np.array(range(N))*dx
y = -L + np.array(range(N))*dy
z = -L + np.array(range(N))*dz
u = np.zeros((N, N, N))
rho = np.zeros((N, N, N))
# source
q = 1.0
rho[(N-1)//2,(N-1)//2,(N-1)//2] = q/(dx*dy*dz)
# préparer l'animation
s = u[:,:,(N-1)//2]
image = plt.imshow(s.T, origin='lower', extent=(-L, L, -L, L), vmax=1.0)
# calculer le paramètre de sur-relaxation
omega = 2.0/(1.0+np.sin(np.pi*dx/L))
# pixels blancs et noirs: les blancs ont i+j+k pairs; les noirs ont i+j+k impairs
blanc = [(i, j, k) for i in range(1, N-1) for j in range(1, N-1) \
         for k in range(1, N-1) if (i+j+k)%2 == 0]
noir = [(i, j, k) for i in range(1, N-1) for j in range(1, N-1) \
         for k in range(1, N-1) if (i+j+k)%2 == 1]
n = 0 # nombre d'itérations
```

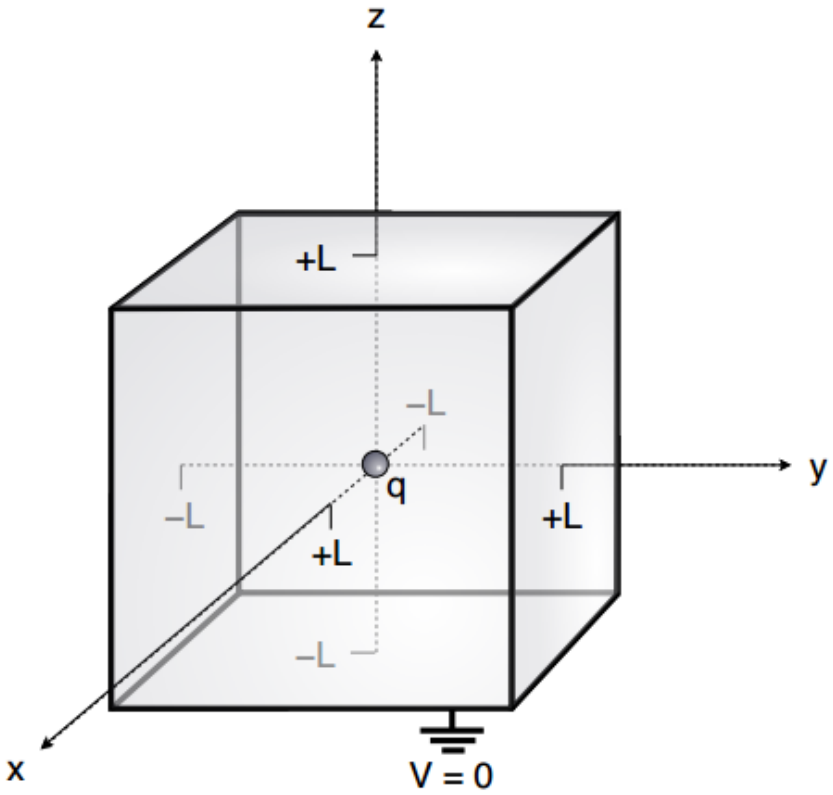


FIGURE 7.7 – Exemple de problème de valeur limite pour l'équation de Poisson : une charge ponctuelle q est située au centre d'un cube de longueur d'arête $2L$ dont les faces sont mises à la terre.

```
err = 1.0 # erreur moyenne par site
while err > eps:
    image.set_data(s.T)
    plt.title('itération %d'%n)
    plt.tight_layout()
    plt.show()
    plt.pause(0.001)
    # prochaine itération en raffinement
    n = n+1
    err = 0.0
    # lboucle sur pixels blancs puis pixels noirs
    for (i, j, k) in blanc+noir:
        du = (u[i-1,j,k] + u[i+1,j,k] + u[i,j-1,k] + u[i,j+1,k] + u[i,j,k-1] \
              + u[i,j,k+1] + dx**2*rho[i,j,k])/6.0 - u[i,j,k]
        u[i,j,k] += omega*du
        err += abs(du)
    err /= N**3
# tracé de surface de la solution finale
(x, y) = np.meshgrid(x, y)
s = s.clip(eps, 1.0)
levels = [10**(l/2.0) for l in range(-5, 0)]
fig = plt.figure()
axis = fig.gca(projection='3d', azim=-60, elev=20)
surf = axis.plot_surface(x, y, s.T, rstride=1, cstride=1, cmap='viridis')
wire = axis.plot_wireframe(x, y, s.T, rstride=1+N//50, cstride=1+N//50,
                           color = "r", linewidth=0.5, alpha = 0.5)
axis.contour(x, y, s.T, levels, zdir='z', offset=-1.0)
axis.contourf(x, y, s.T, 4, zdir='x', offset=-L)
axis.contourf(x, y, s.T, 4, zdir='y', offset=L)
axis.set_zlim(-1.0, 1.0)
```

```
axis.set_xlabel('x')
axis.set_ylabel('y')
axis.set_zlabel('u')
fig.colorbar(surf)
plt.tight_layout()
plt.savefig("charge.png"); plt.savefig("charge.pdf")
plt.show()
```

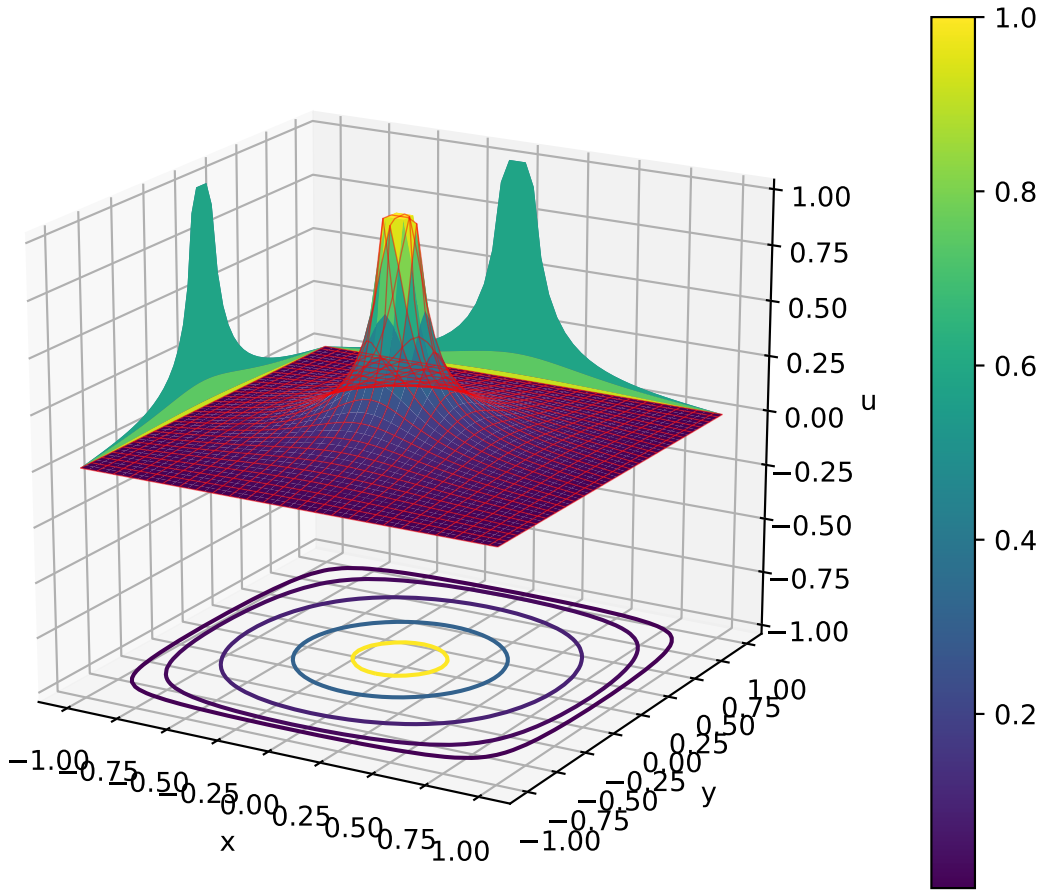


FIGURE 7.8 – Résultats de l'exécution du programme `charge.py` avec des paramètres N = 50 points de grille le long de chaque côté.

7.4 Travaux dirigés

7.4.1 Exercice 1 : Équation de la chaleur 2D

On considère une plaque de fer carrée de côté L = 10 mm, de coefficient de diffusion D = 4 $mm^2.s^{-1}$.

L'équation de diffusion bidimensionnelle est

$$\frac{\partial U}{\partial t} = D\left(\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2}\right) \tag{7.46}$$

Considérons l'équation de diffusion appliquée à une plaque métallique carrée de côté L = 10 mm, de coefficient de diffusion D = 4 $mm^2.s^{-1}$. La plaque est initialement à la température $T_{froid} = 300\ K$ en dehors d'un disque (centré sur $x_c = 5$, y_c =5 et de rayon r = 2 mm) qui est à la température $T_{chaud} = 1000\ K$. Nous supposons que les bords de la plaque sont maintenus fixes à T_{froid}.

a) Rappeler la définition d'un schéma FTCS et donner l'approximation numérique de l'équation de chaleur 2D.

b) Quelle est la valeur du temps maximum Δt que nous pouvons autoriser sans que le processus ne devienne instable ?

c) Terminer lorsqu'il y a des points d'interrogation (???) dans le script Python `EquDiff2D.py` ci-dessous afin d'obtenir la sortie comme indiqué dans la figure 7.9.

```
## NOM DU PROGRAMME: EquDiff2D.py
#% IMPORTATION
import numpy as np
import matplotlib.pyplot as plt
# taille de la plaque, en [mm]
L = 10.
# intervalles dans les directions x, y, en [mm]
dx = dy = 0.1
# Coefficient de diffusion thermique de l'acier, en [mm2.s-1]
D = 4.
Tfroid, Tchaud = 300, 1000
# Nombre de pas de temps
nsteps = 250
nx, ny = int(L/dx), int(L/dy)
dx2, dy2 = dx*dx, dy*dy
# pas de temps maximum, dans question b)
dt = ???
u = Tfroid * np.ones((nx, ny))
# Conditions initiales - disque de rayon r, centrée sur (cx, cy), en [mm]
r, cx, cy = 2, 5, 5
r2 = r**2
for i in range(nx):
    for j in range(ny):
        p2 = ???
        if p2 < r2:
            u[i,j] = Tchaud
# Sortie de 4 graphiques à quatre instants dans l'intervalle de temps
mfig = [0, 50, 100, 200]
fignum = 0 # initialisation
fig = plt.figure()
# implémentation du schéma FTCS
for m in range(nsteps):
```

```
    for i in range(1, nx-1):
        for j in range(1, ny-1):
            uxx = ???
            uyy = ???
            u[i,j] += ???
    if m in mfig:
        fignum += 1
        print(m, fignum)
        ax = fig.add_subplot(220 + fignum)
        im = ax.imshow(u, cmap=plt.get_cmap('hot'), vmin=Tfroid,vmax=Tchaud)
        ax.set_axis_off()
        ax.set_title('{:.1f} ms'.format(m*dt*1000))
fig.subplots_adjust(right=0.85)
cbar_ax = fig.add_axes([0.9, 0.15, 0.03, 0.7])
cbar_ax.set_xlabel('T [K]', labelpad=20)
fig.colorbar(im, cax=cbar_ax)
plt.savefig("EquDiff2D.png"); plt.savefig("EquDiff2D.pdf")
plt.show()
```

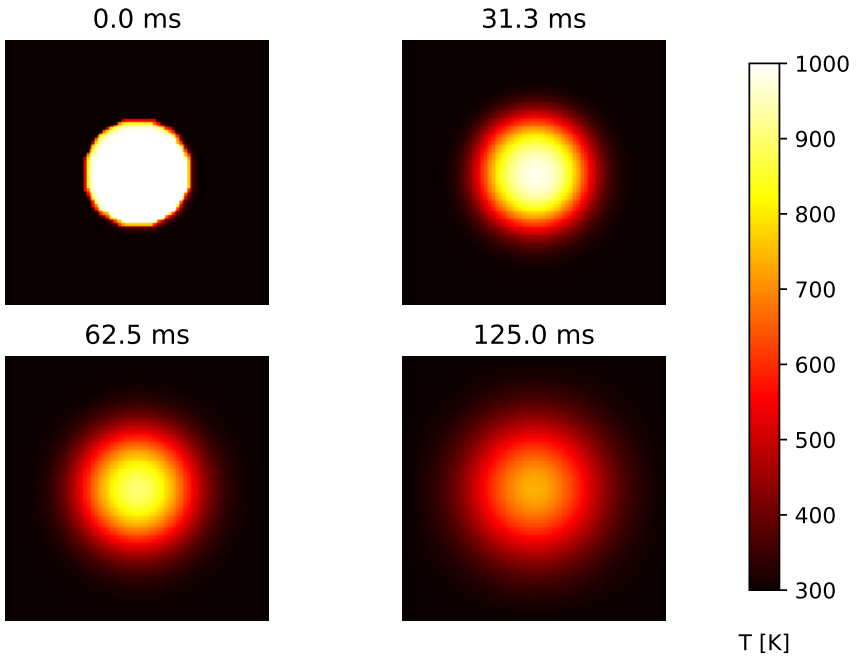


FIGURE 7.9 –

7.4.2 Exercice 2 : Condensateur plan

Résoudre l'équation de Laplace en deux dimensions pour le potentiel dans un condensateur plan

$$\frac{\partial^2 u(x,y)}{\partial x^2} + \frac{\partial^2 u(x,y)}{\partial y^2} = 0 \tag{7.47}$$

Soit un condensateur à plaques parallèles avec des potentiels V = ± 1 V contenus dans une région carrée mise à la terre de longueur latérale L comme indiqué sur la figure 7.10.

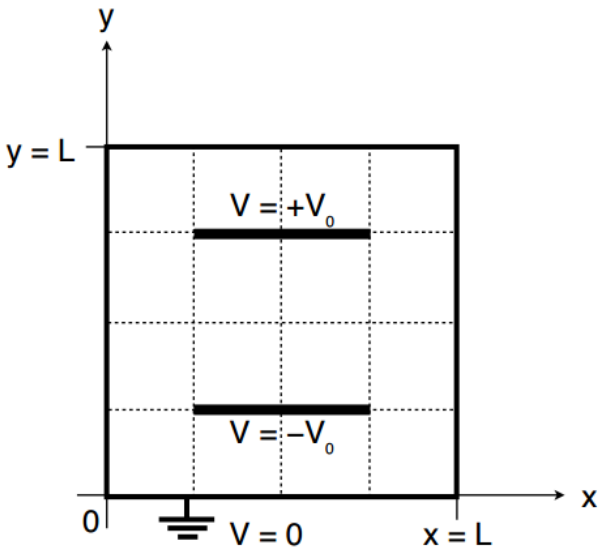


FIGURE 7.10 –

a) Rappeler la définition de la méthode Gauss-Seidel et donner l'approximation numérique de l'équation de Laplace 2D.

b) Adapter le script `Laplace_surrelax.py` étudié dans le cours afin d'avoir les résultats sur les figures 7.11 et 7.12.

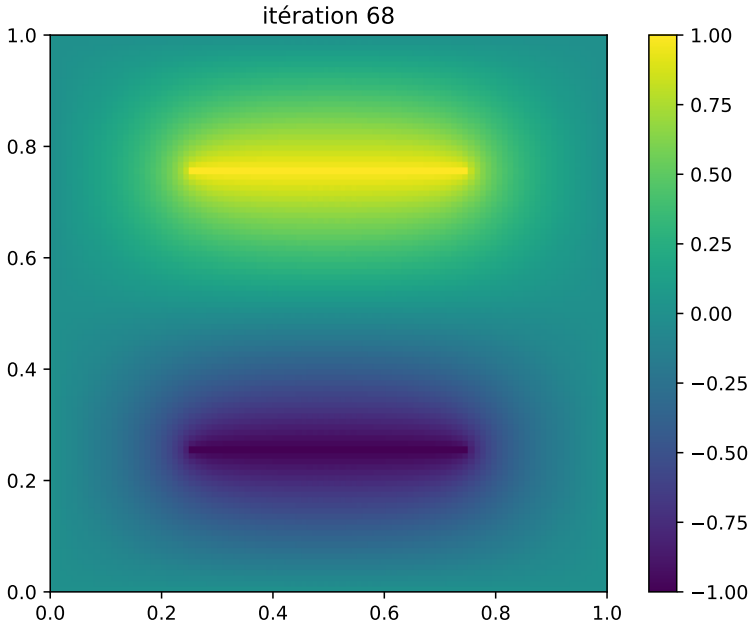


FIGURE 7.11 –

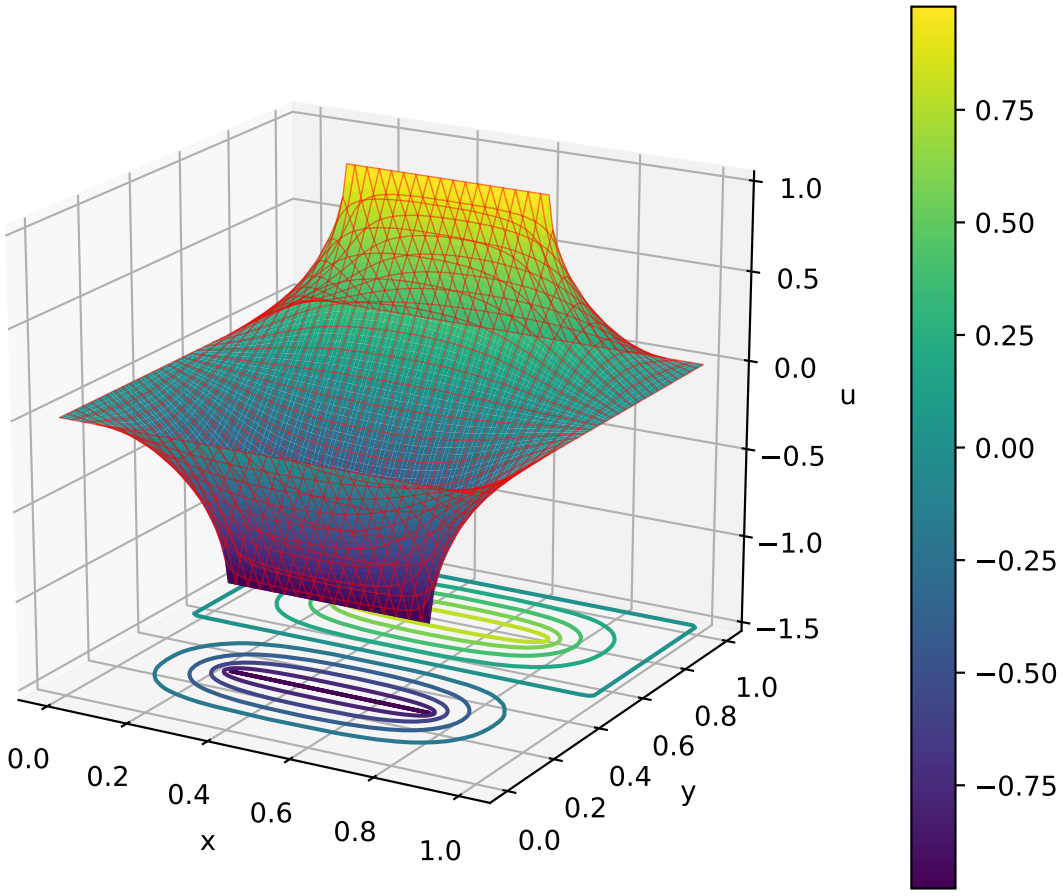

1.0
0.5
0.0
u
−0.5
−1.0
−1.5
0.0
0.2
0.4
0.6
0.8
1.0
x
0.0
0.2
0.4
0.6
0.8
1.0
y
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75

FIGURE 7.12 –