Programmation Python pour les mathématiques

Julien Guillod

Sorbonne Université

2026

Table des matières

Préface

Le but de ce recueil d'exercices est de fournir une introduction à l'usage de la programmation dans divers domaines des mathématiques. Le langage de programmation utilisé est Python 3.6 ou supérieur. Ces exercices servent de base aux travaux pratiques LU2MA100 donnés à Sorbonne Université en licence.

Les exercices de ce recueil correspondent à quelques détails près à ceux de la seconde édition du livre *Programmation Python par la pratique*, publié chez Dunod. Les corrigés sont disponibles dans les versions papier et numérique publiées chez Dunod.

Une traduction en anglais a été publiée sous le titre *Python Programming for Mathematics* par CRC Press.

Pour accéder aux exercices tels qu'ils ont été publiés dans les différentes éditions : 1e édition et 2e édition.

Les notebooks permettant de faire les exercices en dessous des énoncés sont disponibles sur 

et peuvent être lancés directement en ligne sur 

.

1 Introduction

Python est un langage de programmation phare dans le monde scientifique. Il est parfaitement adapté pour programmer des problèmes mathématiques. Cet ouvrage propose de se focaliser sur l'utilisation pratique du langage Python dans différents domaines des mathématiques : les suites, l'algèbre linéaire, l'intégration, la théorie des graphes, la recherche de zéros de fonctions, les probabilités, les statistiques, les équations différentielles, le calcul symbolique, et la théorie des nombres. À travers 55 exercices de difficulté croissante, et corrigés en détails, il permet d'avoir une bonne vision d'ensemble des possibilités d'utilisation de la programmation dans les mathématiques et d'être à même de résoudre des problèmes mathématiques complexes.

Il n'est pas nécessaire de faire les exercices dans l'ordre proposé, même si certains exercices font parfois appel à des notions vues dans des exercices précédents. Les exercices plus difficiles sont indiqués par des points d'exclamation :

— ! : plus long ou plus difficile ;
— !! : passablement long et complexe ;
— !!! : défi proposé sans correction.

Le séparateur décimal utilisé dans cet ouvrage est le point et non pas la virgule, afin de se conformer avec l'usage international employé par Python et éviter les confusions.

1.1 Remerciements

Merci à Marie Postel et Nicolas Lantos pour leurs relectures attentives de la première version de ce recueil et pour les nombreuses corrections et suggestions. Pour cette seconde édition, un merci particulier à Johann Faouzi et Louis Thiry.

Merci également aux membres des équipes pédagogiques de Sorbonne Université ayant utilisé ces exercices pour leurs retours et contributions : Mathieu Barré, Constantin Bône, Jules Bonnard, Cédric Boutillier, Thibault Cimic, Jeanne Decayeux, Cécile Della Valle, Guillaume Duboc, Jean-Jil Duchamps, Jean-Merwan Godon, Elise Grosjean, Cindy Guichard, Nicolas Lantos, Mathieu Mari, David Michel, Leo Miolane, Anouk Nicolopoulos, Arnaud Padrol, Diane Peurichard, Marie Postel, Xavier Poulot-Cazajous, Alexandre Rege, Othmane Safsafi, Emmanuel Schertzer, Agustín Somacal, Didier Smets, Robin Strudel, Gauthier Tallec, Nicolas Thomas, Paul Vernhet, Jules Vidal et Raphaël Zanella.

Finalement merci aux étudiantes et aux étudiants ayant planché sur ces exercices pour leurs retours constructifs qui ont contribué à l'amélioration du présent recueil.

La lectrice attentive ou le lecteur attentif est remercié·e par avance pour tout signalement de coquilles ou autres.

1.2 Pourquoi Python ?

Python est un langage généraliste de programmation interprété qui a la particularité d'être très lisible et pragmatique. Il dispose d'une très grosse base de modules externes, notamment scientifiques, qui le rend particulièrement attractif pour programmer des problèmes mathématiques. Le fait que Python soit un langage interprété le rend plus lent que les langages compilés, il assure en revanche une grande rapidité de développement qui permet à l'humain de travailler un peu moins tandis que l'ordinateur devra travailler un peu plus. Cette particularité fait que Python est devenu l'un des principaux langages de programmation utilisés par les scientifiques.

1.3 Prérequis

Cet ouvrage n'a pas pour but premier d'exposer la syntaxe et les principes du langage Python, aussi les prérequis sont-ils d'en connaître les bases. Il existe de nombreuses ressources pour se mettre à jour en cas de besoin, par exemple :

— le cours en ligne *Python 3 : des fondamentaux aux concepts avancés du langage* d'Arnaud Legout et Thierry Parmentelat à suivre sur le site de *France Université Numérique*. Les vidéos sont également disponibles sur YouTube
— l'ouvrage *Programmation en Python pour les sciences de la vie* de Patrick Fuchs et Pierre Poulain
— le cours *1IN001* dispensé à Sorbonne Université

Par ailleurs la réalisation des exercices demande d'avoir accès à un ordinateur ou un service en ligne disposant de Python 3.6 (ou plus récent) complété par les modules suivants : NumPy, Scipy, SymPy, Matplotlib, Numba, NetworkX et Pandas. L'emploi d'un éditeur de code permettant l'écriture en Python est aussi vivement conseillé. Il est ici suggéré d'utiliser Jupyter Lab, qui permet à la fois l'écriture des notebooks interactifs et des scripts et également l'ajout de ses propres solutions en dessous des énoncés, ce qui est très pratique. Il n'est pas indispensable d'utiliser Jupyter Lab, d'autres environnements sont aussi adaptés, notamment Spyder ou Jupyter Notebook.

Les sections suivantes décrivent comment installer et lancer l'environnement Python ou l'utiliser en ligne sans installation.

1.4 Documentation

Il n'est généralement pas utile (ni souhaitable) de connaître toutes les fonctions et subtilités du langage Python lors d'une utilisation occasionnelle. Par contre il est indispensable de savoir

utiliser la documentation de manière efficace. La documentation officielle est disponible à l'adresse https://docs.python.org/. La langue et la version peuvent être sélectionnées en haut à gauche. Il est fortement conseillé de regarder comment la documentation est écrite et d'apprendre à l'utiliser.

1.5 Installation

Les personnes ne pouvant ou ne voulant pas installer Python peuvent directement se rendre à la Section 1.6 pour des alternatives disponibles en ligne sans installation.

Il existe essentiellement trois façons d'installer Python et les modules requis pour réaliser les exercices :

— *Miniforge* est un gestionnaire de distribution Python. Il faut installer les modules requis pour faire les exercices manuellement. C'est la méthode à privilégier si vous êtes sous Windows ou MacOS.
— *Dépôts Linux :* la plupart des distributions Linux permettent d'installer Python et les modules de base directement à partir des dépôts de paquets qui les accompagnent. C'est la méthode privilégiée sous Linux.
— *Environement virtuel* : permet de gérer plusieurs versions de Python et des modules. Cette méthode permet une gestion plus fine et avancée des modules installés que ce qui est proposé avec les méthodes précédentes.

1.5.1 Installation avec Miniforce

La façon la plus simple d'installer Python 3 et toutes les dépendances nécessaires sous Windows et MacOS est d'installer Miniforge puis d'installer les modules nécessaires manuellement. La documentation détaillée est disponible ici. En résumé la procédure d'installation est la suivante :

1. Télécharger Miniforge depuis : https://conda-forge.org/download/.
2. Double-cliquer sur le fichier téléchargé pour lancer l'installation de Miniforge, puis suivre la procédure d'installation.
3. Une fois l'installation terminée, lancer Miniforge Prompt à partir du menu Démarrer ou de la liste des applications.
4. Dans le terminal, taper la commande :

```
conda install numpy scipy sympy matplotlib numba networkx pandas
 ↪  jupyterlab jupyterlab-lsp python-lsp-server
```

1.5.2 Installation à partir des dépôts

La plupart des distributions Linux permettent d'installer facilement Python et les modules les plus standard directement à partir des dépôts qu'elles incluent. La procédure suivante concerne Ubuntu, mais s'adapte facilement aux autres distributions.

1. Installer Python 3 :

```
sudo apt install python3 python3-pip
```

2. Mettre à jour Pip :

```
pip install --upgrade pip
```

3. Installer les modules NumPy, Scipy, SymPy, Matplotlib, Numba, NetworkX et Pandas :

```
sudo apt install python3-numpy python3-scipy python3-sympy
 ↪  python3-matplotlib python3-numba python3-networkx python3-pandas
```

4. Jupyter Lab n'étant pas disponible dans les paquets d'Ubuntu, il faut l'installer avec Pip :

```
pip install jupyterlab
```

5. De manière optionnelle (mais conseillée), installer l'interface LSP, avec la commande :

```
pip install jupyterlab-lsp python-lsp-server[all]
```

1.5.3 Installation avancée

La procédure suivante décrit l'installation manuelle de modules avec le gestionnaire Pip dans un environnement virtuel.

1. Si Python n'est pas déjà préinstallé sur votre système d'exploitation, installer Python depuis : https://www.python.org/downloads/.
2. Créer un environement virtuel en tapant la commande suivante dans un terminal :

```
python -m venv .venv
```

3. Activer l'environement virtuel, sous Windows en tapant :

```
.venv\Scripts\activate.bat
```

et sous MacOS ou Linux en tapant :

```
source .venv/bin/activate
```

4. Installer les modules requis en tapant la ligne de commande suivante dans un terminal :

```
pip install numpy scipy sympy matplotlib numba networkx pandas
 ↪  jupyterlab jupyterlab-lsp python-lsp-server[all]
```

1.6 Lancement de Jupyter Lab

1.6.1 En ligne de commande

Avec Miniforge, lancer Anaconda Prompt à partir du menu Démarrer ou de la liste des applications. Dans les autres cas, simplement ouvrir un terminal (si un environnement virtuel a été créé, ne pas oublier de l'activer). Ensuite, pour lancer Jupyter Lab en ligne de commande, il faut taper `jupyter lab` dans le terminal. Pour quitter, il faut cliquer sur « Shutdown » dans le menu « File » de la fenêtre Jupyter Lab. Il est aussi possible de taper `Ctrl+C` suivi de `y` (en anglais) ou `o` (en français) dans le terminal où la commande `jupyter lab` a été exécutée.

Remarque : Si le navigateur par défaut est un snap (c'est le cas par défaut sous Ubuntu par exemple), le navigateur renvoie une erreur de fichier non accessible. Pour résoudre ce problème éxécuter la ligne suivante dans un terminal :

```
echo "c.NotebookApp.use_redirect_file = False" >>
 ↪  ~/.jupyter/jupyter_notebook_config.py
```

1.6.2 En ligne sans installation

Pour les personnes ne pouvant ou ne voulant pas installer Python et les dépendances nécessaires sur leur propre ordinateur, il est possible d'utiliser Jupyter Lab en ligne avec launch binder. Aucun compte n'est nécessaire mais les documents modifiés sont automatiquement effacés en quittant donc il faut impérativement les sauvegarder sur votre propre ordinateur avant de quitter. Sinon différents services offrent la possibilité d'utiliser gratuitement Jupyter Lab après création d'un compte :

— CoCalc
— Google Colaboratory

1.7 Utilisation de Jupyter Lab

Une fois Jupyter Lab lancé, la fenêtre représentée à la Figure 1.1 doit apparaître dans un navigateur.

Jupyter Lab permet essentiellement de traiter trois types de documents : les *notebooks*, les *scripts* et les *terminaux*. Un notebook est constitué de cellules qui peuvent contenir soit du code soit du texte au format Markdown. Les cellules de code peuvent être évaluées de manière interactive à la demande, ce qui permet une grande flexibilité. Les cellules de texte peuvent contenir des commentaires, des titres ou des formules LaTeX comme représenté sur la Figure 1.2.

Un script Python est simplement un fichier texte contenant des instructions Python. Il s'exécute en entier de A à Z et il n'est pas possible d'interagir interactivement avec lui pendant son exécution

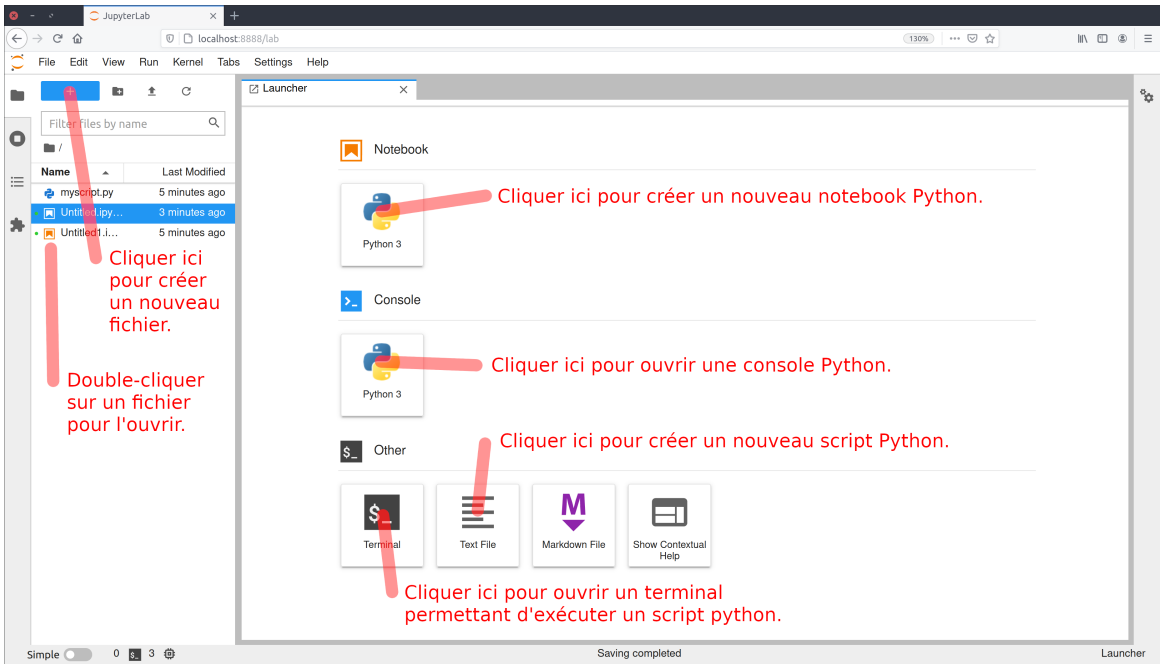


FIGURE 1.1 : Fenêtre de lancement de Jupyter Lab

(à moins que cela n'ait été explicitement programmé). Pour exécuter un script Python il est nécessaire d'ouvrir un terminal.

Commandes de base :

- Créer un nouveau fichier : cliquer sur le bouton « + » situé en haut à gauche, puis choisir le type de fichier à créer.
- Renommer un fichier : cliquer avec le second bouton de la souris sur le titre du notebook (soit dans l'onglet, soit dans la liste des fichiers).
- Changer le type de cellules : menu déroulant permettant de choisir entre « Code » et « Markdown ».
- Exécuter une cellule de code : combinaison des touches `SHIFT+ENTRÉE`.
- Mettre en forme une cellule de texte : combinaison des touches `SHIFT+ENTRÉE`.
- Modifier une cellule de texte : double-cliquer sur la cellule.
- Exécuter un script : taper `python nomduscript.py` dans un terminal pour exécuter le script `nomduscript.py`.
- Réorganiser les cellules : cliquer-déposer.
- Juxtaposer des onglets : cliquer-déposer.

La documentation détaillée de Jupyter Lab est disponible ici.

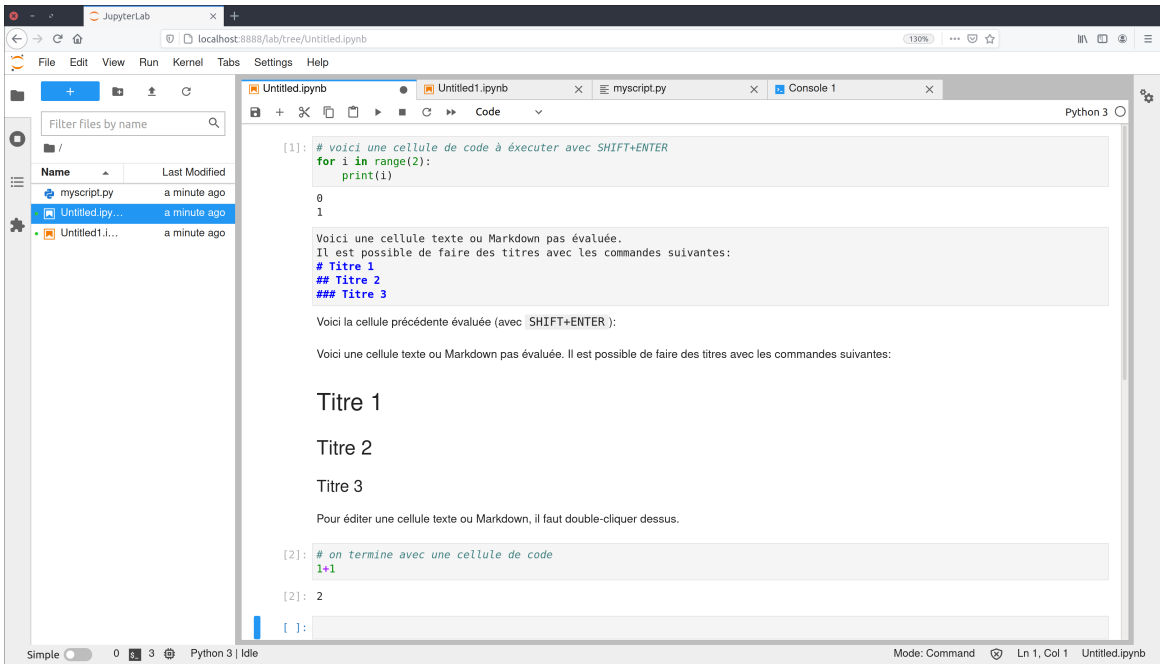


FIGURE 1.2 : Un notebook Jupyter est composé de plusieurs cellules ; ici une cellule de code suivie d'une cellule de texte non évaluée puis la même cellule évaluée et enfin une dernière cellule de code.

1.8 Utilisation avancée de Jupyter Lab

Les versions récentes de Jupyter Lab (3 et suivantes avec un ipykernel supérieur à 6) sont dotées d'un débogueur et d'une interface LSP (Language Server Protocol) particulièrement utiles. Le débogueur permet de trouver des erreurs dans le code en arrêtant le programme à des points particuliers pour comprendre ce qui se passe. La documentation et un tutoriel d'utilisation du débogueur est disponible ici. L'interface LSP permet d'accéder à la documentation et aux signatures des fonctions et offre des diagnostics sur le code ainsi que l'autocomplétion. Les informations d'installation et d'utilisation de l'inteface LSP sont disponibles ici.

Débogueur : En écrivant du code, il est naturel de faire des erreurs, un aspect important est de les localiser et de les identifier de manière efficace. Pour cela, il est possible de mettre des commandes `print` aux bons endroits, mais il est plus approprié d'utiliser un débogueur pour cela. Pour activer le débogueur de Jupyter Lab, cliquez sur le scarabée dans le coin supérieur droit afin qu'il devienne orange. Lorsque le débogueur est activé, la liste des variables globales est disponible dans la barre dédiée. L'aspect le plus utile est la définition de points d'arrêt, qui permettent d'exécuter le code jusqu'à une certaine ligne et d'inspecter l'état du programme à ce moment-là. Pour ce faire, considérons la fonction suivante qui additionne deux nombres :

```
def add(a, b):
    res = a + b
    return res
```

Cliquer à gauche d'un numéro de ligne de code place un point d'arrêt, indiqué par un point rouge. Ici, nous proposons de cliquer sur la deuxième ligne effectuant l'addition. En exécutant le code d'appel de fonction suivant :

```
resultat = add(1, 2)
print(resultat)
```

```
3
```

le programme s'arrêtera à la deuxième ligne de la fonction `add`. Il est possible de visualiser les valeurs des variables `a` et `b` dans l'onglet « Variables » et le code source concerné dans l'onglet « Sources ». Les points d'arrêt sont regroupés dans l'onglet « Breakpoints ». En naviguant dans l'onglet « Callstack », il est possible de poursuivre l'exécution du programme jusqu'au prochain point d'arrêt.

Survol : Lorsque l'on survole le code avec la souris, si une partie devient soulignée, il est alors possible d'obtenir des informations sur la fonction avec la touche `CTRL`. Par exemple, en passant la souris sur le code suivant :

```
from numpy import linalg
```

et en appuyant sur la touche `CTRL` lorsque la souris est sur `numpy` ou `linalg`, une fenêtre avec des explications sur ces modules est affichée. C'est aussi le cas pour les fonctions définies manuellement si elles contiennent une docstring :

```
def square(x):
    """Définition de la fonction puissance x -> x^2"""
    return x*x
```

Déplacer la souris sur le mot `square` :

```
r = square(4)
```

le souligne, et avec la touche `CTRL` la définition apparaît.

Avertissement : Les erreurs ou les avertissements critiques sont indiqués par un soulignement en rouge ou en orange, par exemple dans le cas d'une variable indéfinie :

```
def f(x):
    if x:
        undefined_variable
    return x
```

Suggestion : En tapant `linalg.` dans une cellule, des suggestions de fonctions disponibles dans ce module sont affichées. Dans d'autres cas, les suggestions sont activées avec la touche `TAB`. C'est le cas par exemple avec un dictionnaire défini manuellement :

```
dic = {'key1':3, 'key2':5}
```

En tapant `dic[` dans une cellule suivi de la touche `TAB`, les suggestions `'key1'` et `'key2'` s'affichent.

Signature : En tapant `linalg.solve(`, on obtient l'aide et la signature de cette fonction, c'est-à-dire la façon dont les arguments doivent être utilisés dans cette fonction. En plaçant la souris sur le mot `solve` avec la touche `CTRL`, il y a aussi une description de la fonction.

Référence : En cliquant sur un symbole, les autres utilisations de celui-ci sont mises en évidence.

Définition : En cliquant avec le bouton droit de la souris sur un symbole et ensuite sur « Jump to definition », il est possible d'aller à la définition de la fonction en question. Il est par example possible de faire un test sur le code suivant :

```
f(None)
```

Renommage : Il est possible de renommer une variable de manière intelligente (c'est-à-dire sans renommer les variables locales, par exemple) en faisant un clic droit sur la variable en question et en sélectionnant « Rename symbol ».

Panneau de diagnostic : Il est possible de trier et de naviguer dans les diagnostics à l'aide du « panneau de diagnostic ». Pour l'ouvrir, il suffit de sélectionner « Diagnostics panel » dans le menu contextuel d'une cellule (bouton droit de la souris).

Personnalisation : Le menu « Settings » de Jupyter Lab permet de personnaliser l'environnement de travail, notamment de choisir le thème, la taille de la police, l'indentation par défaut, mais aussi de nombreuses autres options plus avancées.

2 Structures de données

Pour représenter des structures de données, Python propose quatre types de base : les listes (type `list`), les tuples (type `tuple`), les ensembles (type `set`) et les dictionnaires (type `dict`). Le but de ce chapitre est de montrer les différences fondamentales entre ces structures de données et d'expliquer à quoi elles sont le plus adaptées. La documentation détaillée sur les structures de données est disponible ici.

Concepts abordés

- structures de données (liste, tuple, ensemble, dictionnaire)
- types mutable et immutable
- type hashable
- compréhensions de liste, ensemble et dictionnaire
- suites numériques

Exercice 2.1 - Listes

Une liste est une structure permettant de stocker des éléments hétérogènes :

```
list0 = [0, 5.4, "chaîne", True]
```

Les listes sont mutables, c'est-à-dire qu'il est possible d'y modifier un élément, d'en rajouter un ou d'en supprimer un sans avoir à redéfinir toute la liste.

```
list0[3] = False # remplace True par False
list0.append("nouveau") # ajoute la chaîne de caractères "nouveau" à la
 ↪  liste
list0.insert(2, 34) # insère 34 à la place 2
list0.remove(0) # enlève 0
list0
```

```
[5.4, 34, 'chaîne', False, 'nouveau']
```

En particulier, il faut faire attention en copiant une liste. Si l'on exécute le code suivant :

```
list1 = list0
list1[2] = "change"
list0
```

```
[5.4, 34, 'change', False, 'nouveau']
```

alors `list0` est aussi modifié et est égal à `list1`. Pour créer une vraie copie, il faut utiliser le code suivant :

```
list2 = list0.copy()
list2[2] = "rechange"
list0
```

```
[5.4, 34, 'change', False, 'nouveau']
```

qui ne modifie pas `list0`. À noter qu'il est possible de modifier les éléments d'une liste à l'intérieur d'une fonction :

```
def f(l):
    l[0] = 0
f(list0)
list0
```

```
[0, 34, 'change', False, 'nouveau']
```

Enfin il est possible de créer des listes à l'aide de la compréhension de listes :

```
list1 = [2*i+1 for i in range(10)]
list1
```

```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

a. Chercher dans la documentation la syntaxe pour concaténer deux listes.

> **Indication**
>
> Voir la documentation ici.

b. Chercher dans la documentation la syntaxe pour extraire une tranche d'une liste, c'est-à-dire : si a est par exemple une liste de longueur 10, retourner les éléments de 6 à 9.

> **Indication**
>
> Voir la documentation ici.

c. Chercher dans la documentation la syntaxe pour retourner la longueur d'une liste.

d. Écrire une fonction `fibonacci(N)` qui retourne la liste des $N \geq 2$ premiers termes de la suite de Fibonacci définie par $u_{n+2} = u_{n+1} + u_n$ avec $u_0 = 0$ et $u_1 = 1$.

e. Écrire une fonction `pascal(N)` qui retourne la N-ième ligne du triangle de Pascal :

$$
\begin{array}{ccccccccccc}
 & & & & & 1 & & & & & \\
 & & & & 1 & & 1 & & & & \\
 & & & 1 & & 2 & & 1 & & & \\
 & & 1 & & 3 & & 3 & & 1 & & \\
 & 1 & & 4 & & 6 & & 4 & & 1 & \\
1 & & 5 & & 10 & & 10 & & 5 & & 1
\end{array}
$$

f. Soit les suites $(u_n)_{n\in\mathbb{N}}$ et $(v_n)_{n\in\mathbb{N}}$ définies par $u_0 = 1$, $v_0 = 1$, et

$$u_{n+1} = u_n + v_n\,, \qquad\qquad v_{n+1} = 2u_n - v_n\,,$$

pour $n \geq 0$. Calculer u_{100} et v_{100}.

g. Écrire une fonction `vk(n0,K)`, qui pour deux entiers n_0 et $K \geq 1$ calcule la suite des valeurs v_k définies par $v_0 = n_0$ et

$$v_{k+1} = \begin{cases} 3v_k + 1 & \text{si } v_k \text{ est impair,} \\ \frac{v_k}{2} & \text{si } v_k \text{ est pair,} \end{cases}$$

pour $0 \leq k < K$. Pour $K = 1\,000$ et diverses valeurs de $n_0 \in \{10, 100, 1\,000, 10\,000\}$, afficher les cinq dernières valeurs calculées, c'est-à-dire $(v_{K-4}, v_{K-3}, v_{K-2}, v_{K-1}, v_K)$.

Exercice 2.2 - Tuples

Les tuples permettent tout comme les listes de stocker des éléments hétérogènes :

```
tuple0 = (0, 5.4, "chaîne", True)
```

Mais au contraire des listes, les tuples ne sont pas mutables. Il n'est pas possible d'y modifier un élément, d'en rajouter un ou d'en supprimer un sans redéfinir tout le tuple. L'avantage d'un tuple sur une liste est qu'il tuple est hashable, c'est-à-dire qu'il peut être utilisé comme clef dans un dictionnaire.

Enfin il est possible d'affecter des variables à l'intérieur d'un tuple, par exemple :

```
(a,b) = (1,9)
```

Cela est en particulier très utile pour échanger deux variables sans avoir à utiliser une variable supplémentaire :

```
(a,b) = (b,a)
```

a. Vérifier qu'un tuple est bien immutable.

b. Définir une fonction `mdlast(lst,val)` ayant pour argument une liste de tuples d'entiers `lst` et un entier `val` et retourner la liste de tuples avec le dernier élément de chaque tuple remplacé par `val`. Par exemple si `lst = [(10, 20), (30, 40, 50, 60), (70, 80, 90)]` alors `mdlast(lst,100)` doit retourner `[(10, 100), (30, 40, 50, 100), (70, 80, 100)]`.

c. Comment convertir un tuple en liste et réciproquement ?

Exercice 2.3 - Ensembles

Les ensembles permettent de stocker des éléments hétérogènes au sens mathématique de la théorie des ensembles :

```
set0 = {0, 5.4, "chaîne", True}
```

Il est possible de tester si un élément appartient à un ensemble :

```
if "chaîne" in set0:
    print("dedans")
```

```
dedans
```

Les ensembles sont mutables, il est donc possible de rajouter ou retirer un élément d'un ensemble :

```
set0.add(18) # ajoute 18 à l'ensemble
set0.add(0) # ajoute 0 à l'ensemble (ne fait rien car 0 est déjà dedans)
set0.remove("chaîne") # retire "chaîne de l'ensemble
```

En revanche les ensembles ne peuvent contenir que des éléments hashables, c'est-à-dire immutables. En particulier un ensemble ne peut pas contenir un autre ensemble :

```
set1 = { {1,2}, {3}, {4} }
```

```
TypeError: cannot use 'set' as a set element (unhashable type: 'set')
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[23], line 1
----> 1 set1 = { {1,2}, {3}, {4} }
TypeError: cannot use 'set' as a set element (unhashable type: 'set')
```

À noter qu'il existe également en Python des ensembles immutables `frozenset` :

```
frozenset0 = frozenset([0, 5.4, "chaîne", True])
```

Une chaîne de caractères peut être transformée en ensemble :

```
set1 = set('abracadabra')
```

Comme pour les listes, il est possible de faire des compréhensions d'ensembles :

```
set2 = {x for x in 'abracadabra' if x not in 'abc'}
```

Dans cet exemple, les chaînes de caractères sont automatiquement transformées en ensemble. À noter que l'ensemble vide est défini par `set()`.

a. Définir une fonction `divisible(n)` qui retourne l'ensemble des nombres entiers divisibles par `n` inférieurs ou égaux à 100.

b. Chercher dans la documentation comment faire l'intersection, l'union et la différence de deux ensembles. Déterminer les nombres inférieurs ou égaux à 100 qui sont non divisibles par 2 mais divisibles par 3 et 5.

> **Indication**
>
> Voir la documentation de `set` ici.

Exercice 2.4 - Dictionnaires

Les dictionnaires sont une structure permettant de stocker des éléments hétérogènes indexés par des clefs (elles aussi hétérogènes) :

```
dict0 = {"pommes": 0, "poires": 4, 12: 2}
```

Les éléments d'un dictionnaire sont accessibles par les clefs :

```
dict0["pommes"]
dict0[12]
```

2

Un dictionnaire peut être vu comme un tableau associatif associant à chaque clef une valeur. La liste des clefs et celle des valeurs sont accessibles respectivement avec `dict0.keys()` et `dict0.values()`. Les dictionnaires sont mutables, il est donc possible de modifier une association clef-valeur et d'en rajouter ou supprimer une :

```
dict0["pommes"] = 3 # modifie la valeur associée à pommes
dict0["oranges"] = "beaucoup" # rajoute oranges comme clef avec la valeur
 ↪  "beaucoup"
del dict0["poires"] # supprime le couple clef-valeur associé à poires
dict0.pop("pommes") # supprime le couple clef-valeur associé à pommes
```

3

Bien qu'un dictionnaire soit mutable, les clefs qui le composent doivent être des objets hashables, c'est-à-dire immutables. Ainsi une liste ou un ensemble ne peuvent pas servir de clefs dans un dictionnaire :

```
dict0[list0] = "test"
dict0[set0] = "retest"
```

```
TypeError: cannot use 'list' as a dict key (unhashable type: 'list')
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[33], line 1
----> 1 dict0[list0] = "test"
      2 dict0[set0] = "retest"
TypeError: cannot use 'list' as a dict key (unhashable type: 'list')
```

En revanche il est possible d'avoir un tuple ou un frozenset comme clef :

```
dict0[tuple0] = "test"
dict0[frozenset0] = "rest"
```

d'où l'intérêt des frozensets. Comme pour les listes et les ensembles, il est possible de faire des compréhensions de dictionnaires :

```
dict1 = {x: x**2 for x in range(5)}
```

Finalement une chose intéressante avec les dictionnaires est l'unpaking illustré par l'exemple suivant :

```
def add(a=0, b=0):
    return a + b
d = {'a': 2, 'b': 3}
add(**d)
```

```
5
```

a. Comment définir un dictionnaire vide ?

b. Comment concaténer plusieurs dictionnaires entre eux ?

c. On considère une liste de mots :

```
mots = ['Abricot', 'Airelle', 'Ananas', 'Banane', 'Cassis', 'Cerise',
 ↪  'Citron', 'Clémentine', 'Coing', 'Datte', 'Fraise', 'Framboise',
 ↪  'Grenade', 'Groseille', 'Kaki', 'Kiwi', 'Litchi', 'Mandarine',
 ↪  'Mangue', 'Melon', 'Mirabelle', 'Nectarine', 'Orange', 'Pamplemousse',
 ↪  'Papaye', 'Pêche', 'Poire', 'Pomme', 'Prune', 'Raisin']
```

Écrire une fonction `position(mots, x, n)` qui retourne la liste des mots ayant le caractère `x` comme `n`-ième lettre (en commençant à partir de zéro comme en Python).

d. En imaginant que la liste des mots soit très longue, alors à chaque évaluation de la fonction `position` l'ensemble des mots est parcouru, ce qui prend pas mal de temps. Pour améliorer cela, construire un dictionnaire `mots_dict` ayant pour clefs les tuples `(x,n)` et comme valeurs la liste des mots ayant le caractère `x` comme `n`-ième lettre, c'est-à-dire tel que `mots_dict[x,n]` retourne la même chose que `position(mots, x , n)` à l'ordre près. Ainsi la liste `mots` n'est parcourue qu'une seule fois lors de la construction du dictionnaire et ensuite l'évaluation du dictionnaire est extrêmement rapide pour n'importe quelle requête.

3 Structures homogènes

Les structures de données par défaut de Python permettent de gérer des données hétérogènes (par exemple des entiers et des chaînes de caractères). Cette particularité fait que les structures de données Python sont extrêmement flexibles, au détriment de la performance. En effet vu que des données hétérogènes doivent pouvoir être supportées, il n'est pas possible d'allouer une plage de mémoire fixe pour une structure de données, ce qui ralentit son utilisation. Particulièrement en mathématiques, il apparaît très régulièrement des ensembles de données homogènes de tailles fixes (liste d'entiers, vecteurs réels ou complexes, matrices...). Le module NumPy définit le type `ndarray` qui est optimisé pour de telles structures de données homogènes de tailles fixes. La documentation de NumPy est disponible ici.

Pour charger le module NumPy, il est d'usage de procéder ainsi :

```
import numpy as np
```

Concepts abordés

- tableau de données homogènes
- slicing
- opérations vectorielles
- indexage et sélection

Exercice 3.1 - Introduction à NumPy

Création. La taille et le type des éléments d'un tableau NumPy doivent être connus à l'avance. La première façon de créer un tableau NumPy est de construire un tableau rempli de zéros en spécifiant la taille et le type :

```
array0 = np.zeros(3, dtype=int) # vecteur de 3 entiers
array1 = np.zeros((2,4), dtype=float) # tableau de flottants de taille 2x4
array2 = np.zeros((2,2), dtype=complex) # matrice carrée complexe de taille
 ↪  2x2
array3 = np.zeros((5,6,4)) # tableau tridimensionnel de flottants
```

La seconde façon est de passer directement les données :

```
array4 = np.array([1,4,5]) # vecteur d'entiers (1,4,5)
array5 = np.array([[1.1,2.2,3.3,4.4],[1,2,3,4]]) # matrice de taille 2x4 de
 ↪  flottants
```

```
array6 = np.array([[1+1j,0.4],[3,1.5]]) # matrice complexe de taille 2x2
```

NumPy va alors déterminer lui-même le type et la taille du tableau. À noter qu'il est possible de forcer le type :

```
array0 = np.array([1,4,5], dtype=complex) # vecteur de complexes
```

Le type des éléments du tableau NumPy `array1` peut être déterminé par `array1.dtype`. La taille de ce tableau est donnée par `array1.shape`. Les commandes suivantes permettent d'accéder aux éléments des tableaux :

```
array4[1] # retourne 4
array5[1,3] # retourne 4.0
```

```
np.float64(4.0)
```

À noter que les indices commencent à 0 et non pas à 1. Les tableaux NumPy sont mutables dans le sens où les données peuvent être modifiées mais en conservant le même type et la même taille :

```
array0[1] = 4
array1[1,3] = 3.3
array3[3,4,2] = 3
```

Slicing. Le slicing permet d'accéder à certaines parties d'un tableau :

```
array4[2:3] # retourne les éléments d'indices compris entre 2 et 3
array1[0,:] # retourne la première ligne de array1
array1[:,-1] # retourne la dernière colonne de array1
array3[3,3:5,1:4] # retourne la sous-matrice correspondante
```

```
array([[0., 0., 0.],
       [0., 3., 0.]])
```

Itération. Il est possible d'itérer un tableau sur sa première dimension, par exemple pour retourner la somme des lignes :

```
for i in array5:
    print(np.sum(i))
```

```
11.0
10.0
```

a. Étudier la documentation de la fonction `arange` et utiliser cette fonction pour générer les vecteurs $(5, 6, 7, 8, 9)$ et $(3, 5, 7, 9)$s.

> **Indication**
>
> La documentation de la fonction `arange` est disponible ici.

b. Étudier la documentation de la fonction `linspace` et l'utiliser pour générer 10 points équidistribués dans l'intervalle $[2, 5]$.

c. Lire la documentation de la fonction `reshape` et effectuer successivement les transformations suivantes :

$$(1,2,3,4,5,6) \to \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} \to \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \to \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}$$

Exercice 3.2 - Opérations sur les tableaux

Les opérations arithmétiques de base sur les tableaux NumPy sont effectuées élément par élément :

```
mat1 = np.array([[1,2.5,3],[5,6.1,8],[3,2,5]])
mat2 = np.array([[1,0.5,0],[0,0.9,8],[2,0,0]])
mat1 + mat2 # retourne la somme élément par élément
mat1 * mat2 # retourne le produit élément par élément (pas le produit
 ↪  matriciel)
10*mat1**2 # retourne 10 fois le carré des éléments de mat1
```

```
array([[ 10. ,  62.5,  90. ],
       [250. , 372.1, 640. ],
       [ 90. ,  40. , 250. ]])
```

La plupart des fonctions mathématiques définies par NumPy (voir https://numpy.org/doc/stable/reference/routines.math.html) sont également effectuées élément par élément :

```
np.cos(mat1) # retourne le cosinus élément par élément de mat1
np.exp(mat1) # retourne l'exponentielle élément par élément de mat1
```

```
array([[2.71828183e+00, 1.21824940e+01, 2.00855369e+01],
       [1.48413159e+02, 4.45857770e+02, 2.98095799e+03],
       [2.00855369e+01, 7.38905610e+00, 1.48413159e+02]])
```

Le produit matriciel peut être effectué d'une des trois façons suivantes :

```
np.dot(mat1,mat2)
mat1.dot(mat2)
mat1 @ mat2
```

```
array([[ 7.  ,  2.75, 20.  ],
       [21.  ,  7.99, 48.8 ],
       [13.  ,  3.3 , 16.  ]])
```

a. Donné un vecteur $(v_0, v_1, \dots, v_{n-1}) \in \mathbb{R}^n$ la dérivée discrète de ce vecteur est définie par le vecteur $(d_0, d_1, \dots, d_{n-2}) \in \mathbb{R}^{n-1}$ donné par $d_i = v_{i+1} - v_i$ pour $i = 0, 1, \dots, n-2$.

Écrire une fonction `diff_list` qui calcule la dérivée discrète d'une liste et une fonction `diff_np` qui fait la même opération mais sur des vecteurs NumPy en utilisant le slicing.

b. Soit `a_list` et `a_np` respectivement une liste et un tableau de 1 000 éléments tirés au hasard dans l'intervalle $[0, 1]$:

```
a_list = [np.random.random() for _ in range(1000)]
a_np = np.random.random(1000)
```

Comparer le temps d'exécution de `diff_list(a_list)` et de `diff_np(a_np)`.

Indication

Dans Jupyter Lab, il est très facile de déterminer le temps pris par une cellule pour s'évaluer, il suffit de commencer la cellule par `%%time`, par exemple :

```
%%time
result = diff_list(a_list)
```

```
CPU times: user 42  s, sys: 2  s, total: 44  s
Wall time: 47.9  s
```

Pour évaluer la cellule à de multiples reprises et faire une moyenne sur le temps d'exécution afin d'obtenir un résultat plus précis, remplacer `%%time` par `%%timeit`. La documentation est disponible ici.

Exercice 3.3 - Matrice de Vandermonde

Pour $p, n \in \mathbb{N}^*$ et $x = (x_1, \dots, x_p)$ un vecteur de taille p, la matrice de Vandermonde correspondante est définie par :

$$V(x, n) = \begin{pmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} & x_1^n \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} & x_2^n \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & x_{p-1} & x_{p-1}^2 & \cdots & x_{p-1}^{n-1} & x_{p-1}^n \\ 1 & x_p & x_p^2 & \cdots & x_p^{n-1} & x_p^n \end{pmatrix}.$$

a. Écrire une fonction qui construit la matrice $V(x, n)$ élément par élément à l'aide d'une double boucle.

b. Après avoir établi une relation permettant d'écrire la k-ième colonne de $V(x, n)$ uniquement en fonction de x et de k, écrire une deuxième fonction qui construit la matrice $V(x, n)$ colonne par colonne à l'aide de cette relation.

c. Après avoir établi une relation entre la k-ième colonne de $V(x, n)$, sa $(k-1)$-ième colonne et le vecteur x, écrire une troisième fonction qui construit la matrice $V(x, n)$ colonne par colonne à l'aide de cette relation.

d. Comparer les temps d'exécution de ces trois fonctions pour $n = 150$, $p = 100$ et x généré aléatoirement.

Exercice 3.4 - Indexage de tableaux !

Le slicing permet de sélectionner des blocs dans un tableau, mais il est également possible de sélectionner des éléments disparates en utilisant un tableau comme indexage :

```
a = np.arange(12)**2 # tableau des carrés parfaits
i = np.array([1,3,8,5]) # tableau d'indices
a[i] # tableau des éléments de a aux places i
```

```
array([ 1,  9, 64, 25])
```

À noter qu'il est également possible d'indexer par un tableau de dimension supérieure. Le résultat est alors un tableau de la même forme que l'index :

```
j = np.array([[3,4],[9,7]]) # tableau bidimensionnel d'indices
a[j] # sélectionne les éléments de a avec les indices j
```

```
array([[ 9, 16],
       [81, 49]])
```

Pour un tableau à plusieurs dimensions :

```
b = np.array([[0,1,2,3],[4,5,6,7],[8,9,10,11]])
i = np.array([0,1,2,2]) # tableau des premiers indices
j = np.array([1,0,3,1]) # tableau des seconds indices
b[i,j] # sélectionne les éléments d'indices ij
```

```
array([ 1,  4, 11,  9])
```

Enfin il est possible d'indexer un tableau par un tableau de booléens :

```
c = np.array([[0,1,2,3],[4,5,6,7],[8,9,10,11]])
cond = (c >= 5) # tableau de booléens valant True si >= 5 et False sinon
c[cond] = 5 # assigne la valeur 5 à toutes les entrées de c plus grandes
 ↪  que 5
```

Pour la suite, on considère les nombres :

```
[0.9602, -0.99, 0.2837, 0.9602, 0.7539, -0.1455, -0.99, -0.9111, 0.9602,
 ↪  -0.1455, -0.99, 0.5403, -0.99, 0.9602, 0.2837, -0.99, 0.2837, 0.9602]
```

```
[0.9602,
 -0.99,
 0.2837,
 0.9602,
 0.7539,
 -0.1455,
```

```
-0.99,
-0.9111,
0.9602,
-0.1455,
-0.99,
0.5403,
-0.99,
0.9602,
0.2837,
-0.99,
0.2837,
0.9602]
```

comme étant les résultats d'une mesure effectuée toutes les 0.1 seconde aux temps compris entre 2 et 3.7 secondes.

a. Les mesures étant censées être positives, modifier les données pour mettre 0 lorsque les valeurs sont négatives.

b. Calculer les temps pour lesquels les mesures précédentes sont maximales.

c. Pour chaque mesure maximale, retourner la mesure précédente, la mesure maximale et la mesure suivante. Si la mesure précédente ou la suivante n'existent pas, les remplacer par `np.nan`.

4 Représentations graphiques

Représenter graphiquement des résultats issus de calculs numériques ou symboliques est indispensable pour les analyser ou les interpréter. Le module Matplotlib permet de faire des représentations graphiques très variées. La documentation de Matplotlib est disponible ici.

Pour l'utiliser, il est d'usage de l'importer ainsi :

```
import matplotlib.pyplot as plt
```

et souvant NumPy est également très utile :

```
import numpy as np
```

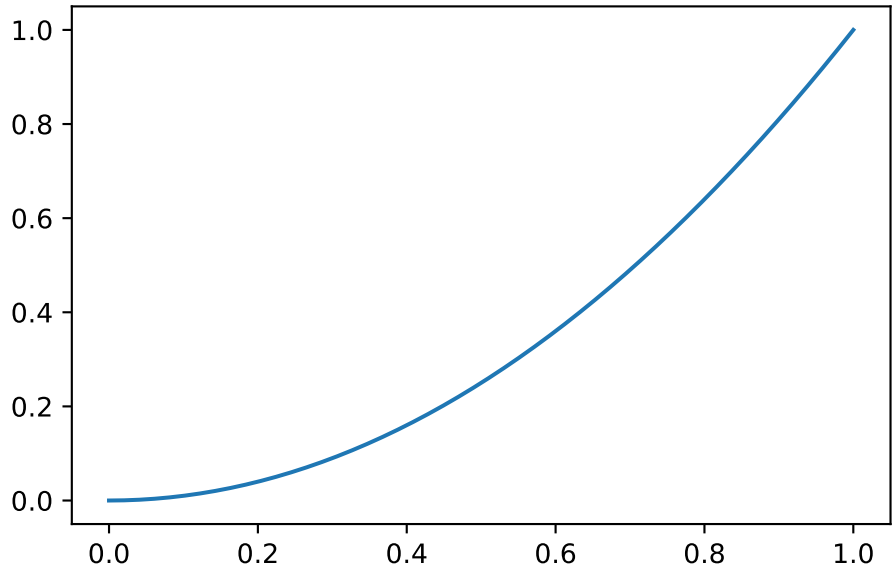
Concepts abordés

- représentation graphique unidimensionnelle
- représentations graphiques bidimensionnelle et tridimensionnelle
- diagramme araignée
- diagramme de bifurcation
- optimisation par parallélisation

Exercice 4.1 - Représentations graphiques

Par exemple la fonction `plot` peut être utilisée pour représenter la fonction x^2 :

```
x = np.linspace(0,1,50)
y = x**2
plt.plot(x,y)
plt.show()
```



Afin de définir une jolie figure pouvant être exportée comme sur la Figure 4.1, la syntaxe est la suivante :

```
plt.figure(figsize=(8,5)) # taille de la figure (en inches)
plt.title(r'Graphique de $x^2$') # titre de la figure (du code LaTeX peut
 ↪  être inclus)
plt.xlabel(r'$x$') # titre de l'axe horizontal
plt.ylabel(r'$y$') # titre de l'axe vertical
plt.plot(x, y, marker='o', label=r"$x^2$") # légende
plt.legend() # affiche la légende
plt.savefig("test.pdf") # exporte la figure en PDF
plt.savefig("test.png", dpi=100) # exporte la figure en PNG
plt.show()
```

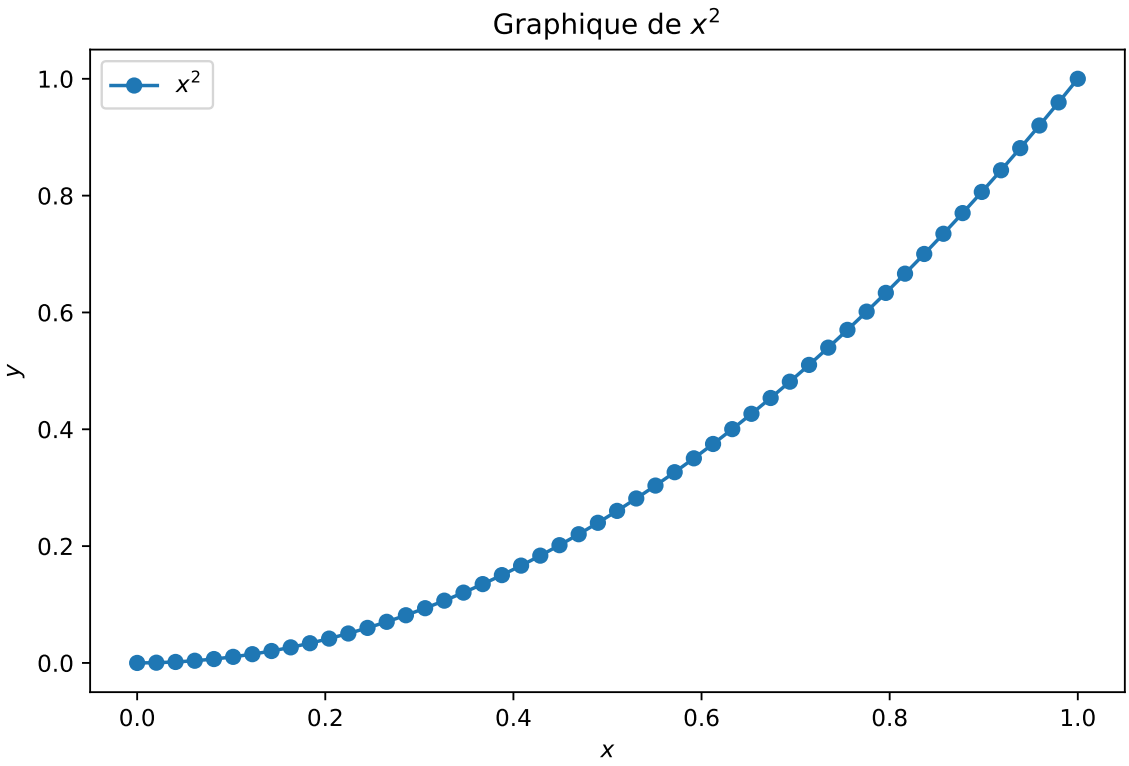


FIGURE 4.1 : Représentation graphique de x^2.

a. Représenter graphiquement sur la même figure les fonctions $\sin(kx)$ et $\cos(kx)$ pour $k = 1, 2, 3$ sur $x \in [0, 2\pi]$. Faire en sorte que les graduations sur l'axe horizontal soient tous les $\frac{\pi}{2}$, comme sur la Figure 4.2.

> **Indication**
>
> Utiliser la fonction `xticks` décrite ici.

b. Regarder l'aide de la fonction `imshow` et l'utiliser pour représenter graphiquement une matrice de nombres aléatoires dans $[0, 1]$ de taille 10×10, comme sur la Figure 4.3.

c. Représenter graphiquement la fonction $f(x, y) = \frac{-y}{5} + e^{-x^2-y^2}$ pour $x \in [-3, 3]$ et $y \in [-3, 3]$ en densité et avec des courbes de niveau, comme sur la Figure 4.4.

Exercice 4.2 - Chaos déterministe

Le but est d'étudier un modèle extrêmement élémentaire d'évolution d'une population. Malgré son caractère simpliste, ce modèle présente de nombreuses propriétés très intéressantes, en particulier chaotiques.

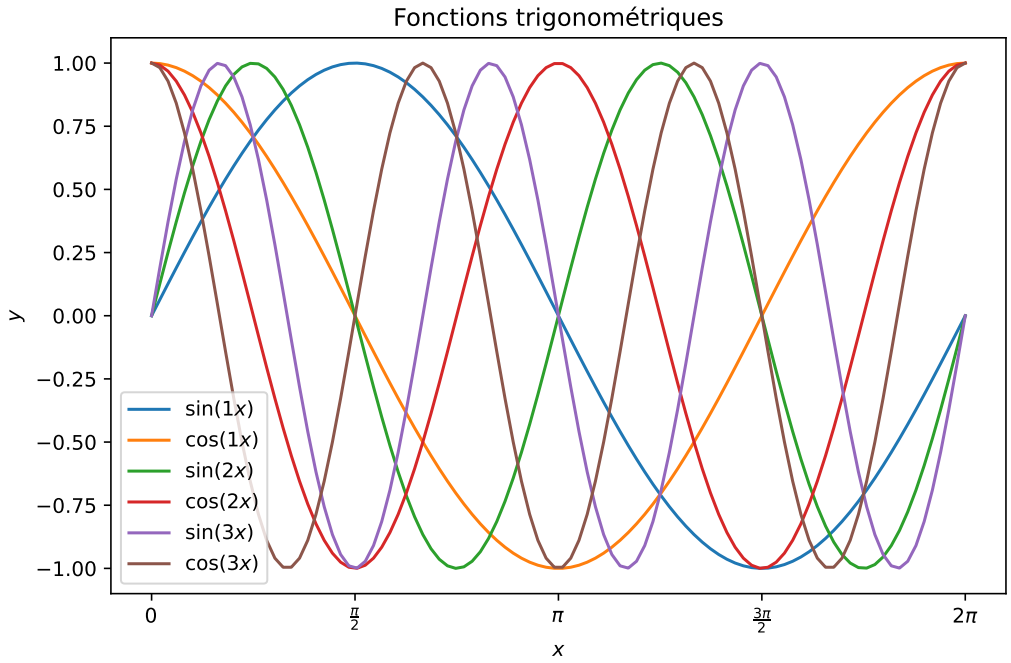


FIGURE 4.2 : Représentation graphique de $\sin(kx)$ et $\cos(kx)$ pour $k = 1, 2, 3$.

La proportion d'une population à un temps discret $i \in \mathbb{N}$ est notée $x_i \in [0,1]$. L'évolution de la population est donnée par $x_{i+1} = f_\mu(x_i)$ où $f_\mu : [0,1] \to [0,1]$ est la fonction définie par :

$$f_\mu(x) = \mu x(1-x)\,.$$

Le paramètre $\mu \in [0,4]$ décrit la croissance de la population. L'application f_μ est appelée l'application logistique de paramètre μ.

a. Définir la fonction f_μ en Python comme $\texttt{f(x, mu)} = f_\mu(x)$ et la représenter graphiquement pour plusieurs valeurs de μ.

b. Pourquoi la valeur de μ ne peut-elle pas être supérieure à 4 dans le modèle ?

c. Définir une fonction qui prend comme argument une valeur de μ, une donnée initiale $x_0 \in [0,1]$ et un nombre $n \in \mathbb{N}$ et retourne la liste $S_\mu^n = (x_0, x_1, x_2, \dots, x_n)$. Modifier cette fonction pour qu'elle puisse prendre un paramètre optionnel $m \in \mathbb{N}$ et retourne la liste $S_\mu^{m,n} = (x_m, x_{m+1}, \dots, x_n)$, c'est-à-dire retire les $m-1$ premiers éléments à S_μ^n.

d. Tester cette fonction en faisant la représentation graphique de la liste S_μ^n pour différentes valeurs des paramètres x_0 et μ. Observer les différents comportements de la suite.

Diagramme araignée. Une façon d'étudier plus spécifiquement ce qui se passe lorsque μ varie est de faire un diagramme dit araignée qui consiste à relier par des droites les points :

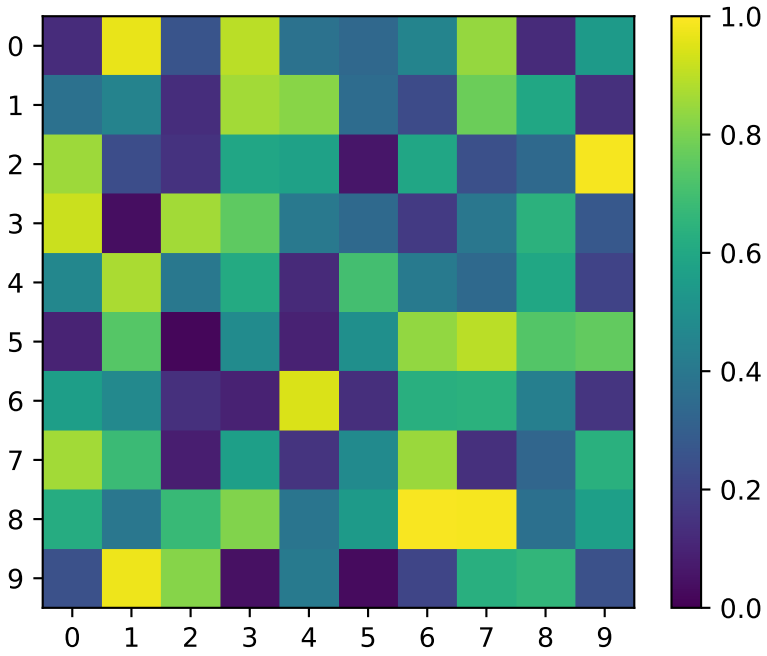


FIGURE 4.3 : Représentation graphique d'une matrice aléatoire. Les couleurs correspondent aux valeurs des entrées de la matrice.

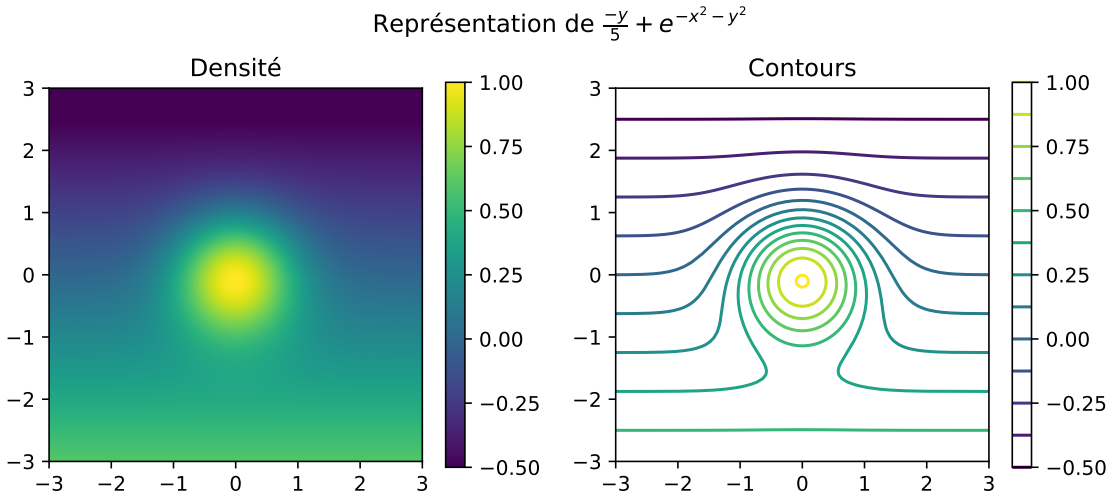


FIGURE 4.4 : Densité et courbes de niveau de la fonction f.

$$\{(x_0, 0), (x_0, x_1), (x_1, x_1), (x_1, x_2), (x_2, x_2), \dots, (x_n, x_n), (x_n, x_{n+1})\}.$$

e. Définir une fonction qui retourne la liste des points nécessaire pour construire le diagramme araignée.

> **Indication**
>
> Dans le but d'utiliser ensuite Matplotlib, il est parfois plus simple de représenter une liste de points $\{(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)\}$ comme le vecteur des abscisses et le vecteur des ordonnées, *i.e.* $(x_0, x_1, \dots, x_n)$ et $(y_0, y_1, \dots, y_n)$.

f. Définir une fonction qui trace le graphe de la fonction f_μ, le graphe de la fonction identité, ainsi que les segments reliant les points de la liste précédente.

g. En expérimentant, étudier qualitativement les effets des paramètres μ et x_0. Décrire le comportement observé lorsque μ augmente.

Diagramme de bifurcation. Les expérimentations précédentes laissent penser que le comportement de la suite x_i en temps grand (*i.e.* lorsque i est grand) est indépendant du choix de la condition initiale x_0 mais dépend beaucoup de la valeur du paramètre μ. Le but de cette section est de représenter graphiquement pour chaque valeur de μ l'ensemble des points $S_\mu^{m,n} = (x_m, x_{m+1}, \dots, x_n)$, *i.e.* en mettant μ sur l'axe horizontal et toutes les valeurs de $S_\mu^{m,n}$ sur l'axe vertical. Un bon choix de paramètres pour nettoyer le diagramme et garder seulement le comportement en temps long du système est $n = 200$ et $m = 100$ par exemple.

h. Définir une fonction qui pour une liste L de valeurs de μ donnée, une donnée initiale $x_0 \in [0, 1]$ et des entiers $m, n \in \mathbb{N}$ retourne la liste des points $\{(\mu, x) : x \in S_\mu^{m,n} \text{ pour } \mu \in L\}$.

i. Définir une fonction qui représente graphiquement cette liste de points. Il est suggéré de prendre pour L une liste de 1 000 valeurs dans l'intervalle $[0, 4]$.

> **Indication**
>
> Utiliser la fonction `scatter` de Matplotlib avec les paramètres `s=1` et `edgecolor='none'`.

j. Interpréter le diagramme obtenu, en particulier ce qu'il dit sur le comportement en temps long du système. Déterminer approximativement pour quelles valeurs de μ le système :

- possède zéro comme unique point fixe ;
- possède un unique point fixe non nul ;
- oscille entre deux valeurs distinctes (cycle de longueur deux) ;
- oscille entre quatre valeurs distinctes (cycle de longueur quatre) ;
- oscille entre trois valeurs distinctes (cycle de longueur trois).

Représentation de l'attracteur. Pour des valeurs de μ proches de 4, les valeurs de la population x_i ont l'air d'être plus ou moins aléatoires. Pourtant, le système est purement déterministe dans le sens où pour une valeur initiale x_0 donnée, la population x_i est définie sans aléa. Ce comportement *a priori* aléatoire est appelé chaos déterministe. Pour bien remarquer que les

points x_i ne sont pas déterminés de manière aléatoire, le but est de représenter graphiquement les points (x_n, x_{n+1}) et de voir que x_{n+1} n'est pas du tout aléatoire par rapport à x_n.

k. Pour chaque valeur de μ fixée, définir une fonction qui retourne la liste de points :

$$\{(x_m, x_{m+1}), (x_{m+1}, x_{m+2}), \dots, (x_n, x_{n+1})\}.$$

l. Représenter graphiquement ces points pour différentes valeurs de μ. Par exemple $n = 5\,000$ et $m = 100$ est un bon choix de paramètres.

m. Comment serait le graphique précédent si chaque x_i était tiré aléatoirement dans l'intervalle $[0, 1]$ indépendamment de x_{i-1} ?

Exercice 4.3 - Ensemble de Mandelbrot

L'ensemble de Mandelbrot est défini comme l'ensemble des points $c \in \mathbb{C}$ pour lesquels la suite de nombres complexes définis récursivement par $z_0 = 0$ et

$$z_{n+1} = z_n^2 + c\,,$$

est bornée. Il est possible de montrer que $c \in \mathbb{C}$ est dans l'ensemble de Mandelbrot si et seulement si $|z_n| \leq 2$ pour tout entier n.

a. Écrire une fonction `mandelbrot(c)` qui vérifie si le point $c \in \mathbb{C}$ est dans l'ensemble de Mandelbrot de manière approximative en testant les cent premières itérations.

b. Tester la fonction précédente avec $c = 0$ et $c = 1 + i$. Qu'est-il attendu théoriquement ?

> **Indication**
>
> Par exemple, le nombre complexe $2 + 3i$ se définit en Python par `2+3j`.

c. Écrire une fonction `mandelbrot_set(N)` qui génère un tableau de taille $N \times N$ représentant l'ensemble $c \in \{x + iy \,:\, x \in [-2, 2] \text{ et } y \in [-2, 2]\}$ et qui retourne un tableau de booléens de taille $N \times N$ déterminant si le point associé est dans l'ensemble de Mandelbrot ou non.

d. En utilisant la fonction précédente avec $N = 100$, représenter graphiquement avec `imshow` une approximation de l'ensemble des points appartenant à l'ensemble de Mandelbrot.

e. En adaptant les fonctions précédentes, représenter graphiquement le logarithme du nombre d'itérations nécessaires avant de ne plus satisfaire $|z_n| \leq 2$ au lieu d'un booléen, comme représenté à la Figure 4.5.

f. !! La méthode précédente a le désavantage de procéder séquentiellement au calcul pour chaque valeur de c, ce qui rend l'évaluation assez lente. Proposer une nouvelle implémentation permettant de calculer parallèlement toutes les valeurs en utilisant les indexages NumPy.

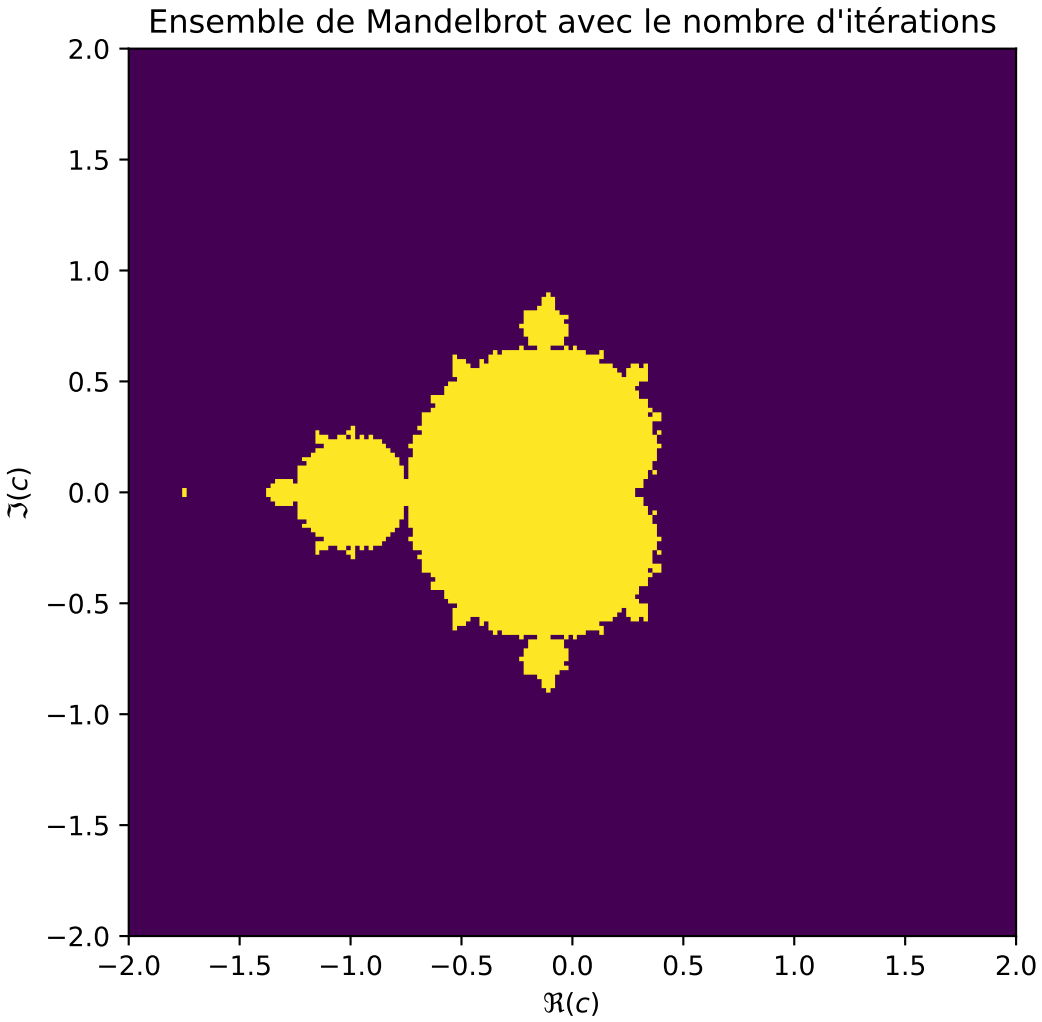


FIGURE 4.5 : Représentation graphique de l'ensemble de Mandelbrot où la couleur représente le logarithme du nombre d'itérations nécessaires pour sortir du disque de diamètre deux.

Exercice 4.4 - Représentations graphiques avancées !

Le but de cet exercice est de découvrir un panel des possibilités offertes par Matplotlib.

a. Tracer les lignes de courant du champ de vecteur de l'oscillateur de Van der Pol :

$$\begin{pmatrix} y \\ -x + \mu(1 - x^2)y \end{pmatrix}$$

pour différentes valeurs de $\mu \in \mathbb{R}$.

b. Représenter graphiquement la courbe paramétrique :

$$\begin{pmatrix} (1 + t^2)\sin(2\pi t) \\ (1 + t^2)\cos(2\pi t) \\ t \end{pmatrix}$$

pour $t \in [-2, 2]$.

c. Représenter la fonction de deux variables :

$$f(x, y) = \sin\left(\sqrt{x^2 + y^2}\right)$$

en trois dimensions pour $x \in [-5, 5]$ et $y \in [-5, 5]$.

d. !! Représenter le ruban de Möbius donné comme surface paramétrique :

$$\begin{pmatrix} (3 + v\cos(u/2))\cos u \\ (3 + v\cos(u/2))\sin u \\ v\sin(u/2) \end{pmatrix}$$

pour $u \in [0, 2\pi]$ et $v \in [-1, 1]$.

e. !! Regarder les exemples disponibles ici et en choisir deux à comprendre et à modifier.

5 Intégration

Le but est d'obtenir une approximation d'une intégrale définie du type

$$J = \int_a^b f(x)\,\mathrm{d}x$$

pour une certaine fonction $f : [a,b] \to \mathbb{R}$ trop compliquée pour *a priori* déterminer la valeur de J à la main. Des méthodes d'approximations déterministes et probabilistes seront introduites pour obtenir une approximation $\tilde{J}$ de J.

Concepts abordés

- méthodes classiques (rectangles, trapèzes et Simpson)
- méthode de Monte-Carlo
- vitesse de convergence
- statistiques

Exercice 5.1 - Méthode des rectangles

La méthode des rectangles est basée sur la définition de l'intégrale au sens de Riemann. La première étape est de découper l'intervalle $[a,b]$ en N intervalles $[x_n, x_{n+1}]$ de même taille $\delta = \frac{b-a}{N}$, *i.e.* $x_n = a + n\delta$ pour $n \in \{0,1,\dots,N-1\}$. La seconde étape consiste à supposer que la fonction f est constante sur chaque intervalle $[x_n, x_{n+1}]$, donc à faire l'approximation

$$J_n = \int_{x_n}^{x_{n+1}} f(x)\,\mathrm{d}x \approx \delta f(\tilde{x}_n)$$

pour $\tilde{x}_n$ une certaine valeur à choisir dans l'intervalle $[x_n, x_{n+1}]$. Le choix de $\tilde{x}_n$ peut par exemple être fait par $\tilde{x}_n = x_n + \alpha\delta$ avec $\alpha \in [0,1]$. Finalement l'approximation de J est donnée par la somme des approximations de J_n,

$$\tilde{J} = \sum_{n=0}^{N-1} \delta f(\tilde{x}_n)\,.$$

En supposant que $f \in C^1([a,b])$, alors il est possible de montrer que la méthode des rectangles converge et que sa vitesse de convergence est d'ordre un. Une méthode numérique est dite d'ordre k si l'erreur entre l'approximation numérique et le résultat exact est de l'ordre de N^{-k}.

a. Choisir une fonction continue $f : [a,b] \to \mathbb{R}$ et définir la fonction Python `f(x)` correspondante. Pour tester le code, il est judicieux de choisir une fonction f dont l'intégrale peut être facilement calculable à la main.

> **Indication**
>
> La liste des fonctions mathématiques de base disponibles en Python dans le module `math` est disponible ici. À noter que NumPy définit également des fonctions mathématiques, voir la documentation ici.

b. Écrire une fonction `rectangles(f,a,b,N)` qui retourne l'approximation de l'intégrale J par la méthode des rectangles par exemple en choisissant $\tilde{x}_n = x_n$, *i.e.* le bord gauche de l'intervalle $[x_n, x_{n+1}]$.

> **Indication**
>
> Il n'est pas nécessaire de stoker toutes les valeurs des approximations de J_n, mais il est possible d'incrémenter une variable pour chaque approximation de J_n.

c. Modifier la fonction précédente pour que celle-ci prenne un paramètre optionnel `alpha` déterminant le choix du paramètre $\alpha \in [0,1]$.

d. Écrire une fonction `plot_rectangles(f,a,b,N,alpha=0.5)` qui représente graphiquement l'approximation par la méthode des rectangles.

e. Déterminer empiriquement la vitesse de convergence de la méthode des rectangles en fonction de N.

f. Déterminer analytiquement la convergence de la méthode des rectangles. Quelles sont les hypothèses nécessaires sur f ?

> **Indication**
>
> Utiliser le théorème des accroissements finis.

Exercice 5.2 - Méthode des trapèzes

La méthode des trapèzes est basée sur une approximation linéaire sur chaque intervalle $[x_n, x_{n+1}]$, plus spécifiquement :

$$J_n = \int_{x_n}^{x_{n+1}} f(x)\,\mathrm{d}x \approx \delta \frac{f(x_n) + f(x_{n+1})}{2}.$$

a. Écrire une fonction Python `trapezes(f,a,b,N)` qui retourne l'approximation de l'intégrale J par la méthode des trapèzes. Tester la fonction `trapezes(f,a,b,N)` pour différentes fonctions f.

b. L'implémentation de votre fonction `trapezes(f,a,b,N)` est-elle optimale quant au nombre d'évaluations de f effectuées par rapport au nombre d'évaluations nécessaires? Une implémentation optimale de la fonction `trapezes(f,a,b,N)` devrait effectuer $N+1$ évaluations de f.

c. Déterminer empiriquement la vitesse de convergence de la méthode des trapèzes en fonction de N.

d. ! Déterminer analytiquement la convergence de la méthode des trapèzes. Quelles sont les hypothèses nécessaires sur f ?

Exercice 5.3 - Méthode de Monte-Carlo

La méthode de Monte-Carlo (du nom des casinos, pas d'une personne) est une approche probabiliste permettant d'approximer la valeur d'une intégrale. L'idée de base est que l'intégrale J peut être vue comme l'espérance d'une variable aléatoire uniforme X sur l'intervalle $[a,b]$:

$$J = \int_a^b f(x)\,\mathrm{d}x = (b-a)\mathbb{E}(f(X))\,.$$

Par la loi des grands nombres cette espérance peut être approximée par la moyenne empirique :

$$\tilde{J} = \frac{b-a}{N}\sum_{i=0}^{N-1} f(x_i)\,,$$

où les x_i sont tirés aléatoirement dans l'intervalle $[a,b]$ avec une loi de probabilité uniforme.

a. Écrire une fonction `montecarlo(f,a,b,N)` qui détermine une approximation $\tilde{J}$ de J par la méthode de Monte-Carlo.

> **Indication**
>
> Pour générer un vecteur de nombres aléatoires, le sous-module `random` de NumPy peut être utile, voir la documentation ici.

b. Modifier la fonction précédente, pour qu'elle retourne en plus de la moyenne $\tilde{J}$ également la variance empirique :

$$\tilde{V} = \frac{(b-a)^2}{N}\sum_{i=0}^{N-1}\left(f(x_i) - \frac{\tilde{J}}{b-a}\right)^2\,.$$

c. Étudier empiriquement la convergence de la méthode de Monte-Carlo en fonction de N en faisant pour chaque valeur de N une statistique sur k exécutions. Plus précisément cela consiste à faire k évaluations de $\tilde{J}$ par la méthode de Monte-Carlo et de calculer la moyenne et la variance des k résultats obtenus.

d. Déterminer analytiquement la convergence de la méthode de Monte-Carlo. Quelles sont les hypothèses nécessaires sur f ?

> **Indication**
>
> Utiliser le théorème central limite.

Exercice 5.4 - Méthode de Simpson !

La méthode de Simpson consiste à approximer la fonction f sur chaque intervalle $[x_n, x_{n+1}]$ par un polynôme de degré deux. Le choix le plus naturel est le polynôme p_n de degré deux passant par les points $(x_n, f(x_n))$, $(\frac{x_n+x_{n+1}}{2}, f(\frac{x_n+x_{n+1}}{2}))$ et $(x_{n+1}, f(x_{n+1}))$.

a. Déterminer la forme explicite du polynôme p_n.

> **Indication**
>
> Le polynôme $L(x) = \frac{(x-c)(x-b)}{(a-c)(a-b)}$ prend la valeur 1 en $x = a$ et la valeur 0 en $x = b$ et $x = c$. Faire une combinaison linéaire de trois polynômes de ce type.

b. Calculer l'approximation donnée par $J_n \approx \int_{x_n}^{x_{n+1}} p_n(x)\,\mathrm{d}x$.

> **Indication**
>
> Il est possible de calculer cette intégrale à la main ou bien de le faire avec le module SymPy, voir la documentation ici.

c. Simplifier à la main la somme $\tilde{J}$ des approximations de J_n.

d. Écrire une fonction `simpson(f,a,b,N)` permettant de calculer une approximation de J avec la méthode de Simpson.

e. Comparer la précision, c'est-à-dire la vitesse de convergence, des méthodes des rectangles, des trapèzes et de Simpson en fonction de N pour une fonction lisse et la fonction $f(x) = \sqrt{1-x^2}$ sur $[0, 1]$ (dont l'intégrale est $\frac{\pi}{4}$).

f. Si ce n'est pas déjà fait, proposer une implémentation parallèle de la méthode de Simpson en utilisant l'indexation NumPy.

Exercice 5.5 - Intégration avec Scipy !!

Les méthodes d'intégrations précédentes et d'autres sont définies dans le module `integrate` de Scipy. Ce module permet en particulier de traiter des cas plus compliqués : intégrales singulières, généralisées ou en plusieurs dimensions.

a. Définir une fonction `E(n,x)` calculant numériquement l'intégrale suivante :

$$E_n(x) = \int_1^\infty \frac{e^{-xt}}{t^n} \mathrm{d}t \,.$$

> **Indication**
>
> Lire la documentation du sous-module `integrate` de Scipy disponible ici.

b. Déterminer une approximation de l'intégrale double :

$$I = \int_0^\pi \left(\int_0^y x \sin(xy) \, \mathrm{d}x \right) \mathrm{d}y \,.$$

6 Algèbre

Tout d'abord une méthode de résolution de système linéaire par un algorithme direct est étudiée, puis une méthode itérative sera utilisée pour déterminer le vecteur propre associé à la plus grande valeur propre d'une matrice. Enfin les groupes générés par un ensemble de permutations seront étudiés.

Concepts abordés

- méthode de résolution directe (décomposition LU)
- algorithme *in place*
- méthode itérative (puissance itérée)
- groupes de permutations
- orbite et stabilisateur

Exercice 6.1 - Décomposition LU

La décomposition LU consiste à décomposer une matrice A de taille $n \times n$ sous la forme $A = LU$ où L est une matrice triangulaire inférieure avec des 1 sur la diagonale et U une matrice triangulaire supérieure. Une fois la décomposition $A = LU$ d'une matrice connue, il est alors très facile de résoudre le problème linéaire $Ax = b$ en résolvant d'abord $Ly = b$ puis $Ux = y$. L'avantage de la décomposition LU sur l'algorithme de Gauss, par exemple, est qu'une fois la décomposition LU connue, il est possible de résoudre le système linéaire rapidement avec des seconds membres différents.

Vu que $l_{ik} = 0$ si $k > i$, nous avons :

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

et donc comme $l_{ii} = 1$:

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj} \,.$$

D'un autre côté, vu que $u_{kj} = 0$ si $k > j$, alors :

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

et donc si $u_{jj} \neq 0$:

$$l_{ij} = \frac{1}{u_{jj}}\left(a_{ij} - \sum_{k=1}^{j-1} l_{ik}u_{kj}\right).$$

Ainsi, si les $(i-1)$ premières lignes de U et les $(i-1)$ premières colonnes de L sont connues, il est possible de déterminer la i-ème ligne de U par :

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik}u_{kj}\,, \quad j \geq i\,,$$

puis la i-ème colonne de L par :

$$l_{ji} = \frac{1}{u_{ii}}\left(a_{ji} - \sum_{k=1}^{i-1} l_{jk}u_{ki}\right), \quad j > i\ .$$

Pour que la décomposition LU d'une matrice A soit possible il faut que les u_{ii} ne soient jamais nuls. Cela est effectivement le cas lorsque la matrice A et toutes ses sous-matrices principales sont inversibles.

a. Écrire une fonction `LU(A)` qui retourne le résultat de la décomposition LU d'une matrice.

b. Donnée la décomposition LU d'une matrice A, écrire une fonction `solve(L,U,b)` qui résout le système linéaire $Ax = b$.

c. Écrire une fonction `LU_inplace(A)` qui ne crée pas de nouveaux tableaux pour L et U mais modifie A de sorte que sa partie triangulaire inférieure (sans la diagonale) corresponde à L et sa partie triangulaire supérieure (avec la diagonale) corresponde à U. Faire également une version `solve_inplace` prenant en entrée la sortie de `LU_inplace` et retournant la solution x, sans utiliser de nouveaux tableaux.

d. En utilisant la décomposition LU de la matrice A, écrire une fonction `det(A)` qui retourne son déterminant.

Exercice 6.2 - Méthode de la puissance itérée

Le but de cet exercice est de déterminer le vecteur propre d'une matrice associé à la valeur propre de plus grand module, en supposant que celle-ci est unique. Étant donné une matrice réelle A de taille $n \times n$ et un vecteur $x_0 \in \mathbb{R}^n$, la suite de vecteurs $(x_k)_{k\in\mathbb{N}}$ est définie par :

$$x_{k+1} = \frac{Ax_k}{\|Ax_k\|}.$$

En supposant par exemple que la matrice A soit diagonalisable, alors il est possible de montrer que la suite $(x_k)_{k\in\mathbb{N}}$ converge à un signe près vers le vecteur propre associé à la plus grande valeur propre de A en valeur absolue. La convergence a lieu presque sûrement pour tous les choix de x_0.

a. Définir une fonction `puissance(A, x0, k)` qui retourne x_k. Avec cette fonction, déterminer le plus grand vecteur propre de la matrice :

$$A = \begin{pmatrix} 0.5 & 0.5 \\ 0.2 & 0.8 \end{pmatrix}.$$

b. Déterminer la valeur propre associée au vecteur propre précédent.

> **Indication**
>
> Si v est un vecteur propre normalisé de A, alors la valeur propre associée est donnée par $\lambda = Av \cdot v$.

c. Écrire un programme permettant de calculer automatiquement la valeur propre de plus grand module et le vecteur propre associé d'une matrice carrée avec une certaine précision donnée. On choisira le vecteur initial x_0 aléatoirement.

d. En supposant la matrice A diagonalisable avec une unique valeur propre de plus grand module, montrer que la suite x_k converge à un signe près vers le vecteur propre associé à cette valeur propre de plus grand module pour presque tous les choix de x_0.

> **Indication**
>
> Décomposer x_0 dans la base des vecteurs propres de A. Pour simplifier, on pourra supposer que la valeur propre de plus grand module est positive.

e. Regarder la documentation de NumPy pour trouver les fonctions permettant de calculer les vecteurs propres et valeurs propres d'une matrice.

f. Comparer la rapidité du code précédent et des fonctions NumPy pour des matrices de tailles $n \times n$ avec $n = 10, 100, 1\,000$.

Indication

En prenant par exemple des matrices dont les coefficients sont générés aléatoirement dans l'intervalle $(0, 1)$, le théorème de Perron-Frobenius assure l'existence d'une unique valeur propre de module maximal et celle-ci est positive.

Exercice 6.3 - Exponentielle de matrices

Le but de cet exercice est de développer un algorithme permettant de calculer l'exponentielle d'une matrice réelle carrée. Si A est une matrice réelle carrée, son exponentielle est définie par la série :

$$\exp(A) = \sum_{k=0}^{\infty} \frac{A^k}{k!},$$

par analogie avec l'exponentielle sur les nombres réels. Ici A^k représente le produit matriciel de A avec elle-même k fois.

a. Définir les tableaux NumPy correspondant aux matrices A_1 et A_2 définies par :

$$A_1 = \begin{pmatrix} 1 & 0.8 & 0.6 \\ 0.8 & 0.2 & 0.8 \\ 0 & 1.2 & 0.9 \end{pmatrix}, \qquad A_2 = \begin{pmatrix} 2 & 3 & 2 \\ 1 & 2 & 3 \\ 4 & 3 & 5.2 \end{pmatrix}.$$

b. Définir une fonction `matrix_power(A,n=20)` retournant une approximation de $\exp(A)$ obtenue en gardant uniquement les $n + 1$ premiers termes de la série, *i.e.* les termes de $k = 0$ à $k = n$.

c. Tester sur les matrices A_1 et A_2 et comparer avec les résultats de la fonction `expm` du module `linalg` de Scipy.

d. Sans utiliser la fonction `norm` de NumPy ou Scipy, définir une fonction calculant la norme de Frobenius $\|A\|_F$ d'une matrice A de taille $m \times m$ définie par :

$$\|A\|_F^2 = \mathrm{tr}(AA^t) = \sum_{i=1}^{m} \sum_{j=1}^{m} |a_{ij}|^2 .$$

Calculer les normes de Frobenius des matrices A_1 et A_2.

e. Pour les matrices A_1 et A_2, tracer l'erreur en norme de Frobenius entre le résultat de `matrix_power` et le résultat de `expm` en fonction du nombre de termes n gardés. Mettre une échelle logarithmique sur l'axe des ordonnées.

D'un point de vue théorique, il est possible de montrer que l'erreur est bornée par :

$$\left\| \exp(A) - \sum_{k=0}^{n} \frac{A^k}{k!} \right\|_F \leq \frac{e^{\|A\|_F}}{(n+1)!} \|A\|_F^{n+1} .$$

f. Tracer cette borne en fonction de n pour différentes valeurs de la norme $\|A\|_F$ allant de 2 à 20, avec également une échelle logarithmique sur les ordonnées. En déduire grossièrement le nombre de termes à garder pour que la borne soit inférieure à la précision machine de 10^{-15} si $\|A\|_F = 20$. Comparer la borne théorique avec ce qui a été observé à la question précédente.

Une idée basique pour améliorer la convergence de la série lorsque la norme de la matrice est grande est d'effectuer un changement d'échelle à l'aide de la relation :

$$\exp A = \left(\exp(A/p)\right)^p .$$

pour $p \geq 1$ un entier bien choisi tel que $\|A/p\|_F$ soit petite, par exemple inférieure à un.

g. En utilisant la propriété précédente, écrire une fonction `matrix_power_opt(A,n=20)` fondée sur cette propriété.

h. Refaire le même graphique qu'au point *e* mais avec cette nouvelle fonction et commenter.

Exercice 6.4 - Groupes de permutations

Le but est d'étudier les groupes de permutations en développant des algorithmes pour les caractériser. Un groupe de permutations $G \subset S_n$ est défini comme étant généré par un certain nombre de permutations : $G = \langle g_1, g_2, \dots, g_k \rangle$, avec $g_i \in S_n$ une permutation de l'ensemble $\{1, 2, \dots, n\}$. Une permutation

$$g = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 1 & 2 & 3 \end{pmatrix},$$

peut être représentée en Python par le tuple `g = (0, 4, 1, 2, 3)`. Le zéro est ajouté afin de se conformer avec l'indexation à partir de zéro de Python et ainsi de simplifier un peu les implémentations. Cela veut simplement dire que le sommet `0` va sur le sommet `0`. À noter que cet exercice se prête particulièrement bien à une implémentation orientée objet, et dans ce cas les questions peuvent être adaptées en conséquence.

a. Écrire une fonction `product(g1,g2)` qui calcule le produit de deux permutations.

b. Écrire une fonction `inverse(g)` qui calcule l'inverse d'une permutation.

c. Écrire une fonction qui retourne la décomposition d'une permutation sous forme de cycles représentés par une liste du tuples.

d. Écrire une fonction qui retourne la permutation correspondant à une liste de cycles, c'est-à-dire qui fait l'inverse de la fonction précédente.

e. ! En Python un groupe $G = \langle g_1, g_2, \dots, g_k \rangle$ engendré par une famille de permutations, peut être représenté par une liste de permutations `G = [g1,g2,...,gk]`. Écrire une fonction `orbit(G,x)` qui retourne l'orbite d'un point $x \in \{1, 2, \dots, n\}$:

$$O_x = Gx = \{gx, g \in G\}.$$

Indication

Ne pas calculer l'ensemble des éléments du groupe, cela fait une liste beaucoup trop longue. Construire la liste $(X^0, X^1, \dots, X^N)$ d'ensembles disjoints définie récursivement par $X^0 = \{x\}$ et X^n comme l'ensemble des éléments nouveaux de la forme $g_i y$ avec $1 \leq i \leq k$ et $y \in X^{n-1}$:

$$X^n = \left(\bigcup_{i=1}^{k} g_i X^{n-1} \right) \setminus \left(\bigcup_{i=1}^{n-1} X^i \right).$$

f. !! Définir une fonction `stabilizer(G,x)` qui retourne le stabilisateur d'un point $x \in \{1, 2, \dots, n\}$:

$$G_x = \{g \in G : gx = x\},$$

sous la forme d'un ensemble de générateurs.

Indication

Utiliser le théorème ou lemme de Schreier.

7 Théorie des graphes

Un graphe est un couple $G = (X, E)$ constitué d'un ensemble X, non vide et fini, et d'un ensemble E de paires d'éléments de X. Les éléments de X sont les sommets du graphe G, ceux de E sont les arêtes du graphe G. Un graphe est orienté si les arêtes ont une direction, c'est-à-dire si les couples d'éléments de E sont des listes ordonnées telles que $(i, j) \in E$ n'est pas équivalent à $(j, i) \in E$. Ici seuls des graphes non orientés, c'est-à-dire dont les paires d'éléments de X sont des ensembles non ordonnés ($\{i, j\} \in E$), sont considérés.

Par exemple, le graphe complet à n sommets K_n est le graphe de sommets $X = \{1, 2, \dots, n\}$ ayant comme arêtes les parties à deux éléments de X. En particulier, $K_4 = (X, E)$ où $X = \{1, 2, 3, 4\}$ et $E = \big\{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}, \{3, 4\}\big\}$.

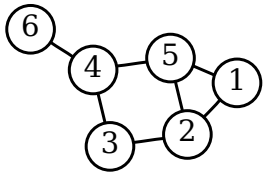
Concepts abordés

- graphes non orientés
- représentation comme dictionnaire
- utilisation des frozensets
- matrice d'adjacence
- recherche de chemins et de triangles
- fonction récursive

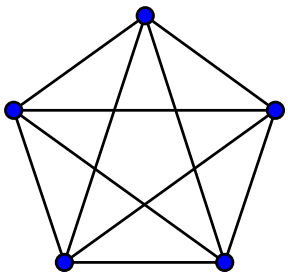
Exercice 7.1 - Graphes comme dictionnaires

Une façon de représenter un graphe G est de le faire avec un dictionnaire dont les clefs sont les sommets et la valeur associée à chaque clef $x \in X$ est un ensemble contenant les voisins de x.

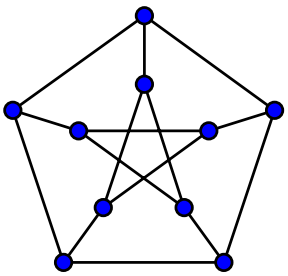
a. Construire sous forme de dictionnaire les graphes suivants :



(a) Graphe à six sommets



(b) Graphe complet



(c) Graphe de Petersen

b. Écrire une fonction `complet(n)` qui construit comme dictionnaire le graphe complet K_n.

c. Un graphe donné sous forme de dictionnaire contient l'information plusieurs fois. Écrire une fonction `corriger(graphe)` permettant de rajouter les éléments manquants de sorte que pour tout sommet `x`, si `y` appartient à `graphe[x]`, alors `y` est aussi une clef et `x` appartient à `graphe[y]`. Tester cette fonction.

d. Écrire une fonction qui retourne l'ensemble (type `set`) de toutes les arêtes d'un graphe représenté par un dictionnaire.

> **Indication**
>
> Les ensembles sont mutables et donc pas hashables.

e. ! Écrire une fonction permettant de déterminer si deux sommets sont connectés par un chemin ou non et qui retourne le chemin si oui.

> **Indication**
>
> Utiliser une fonction récursive.

f. ! Écrire une fonction qui retourne tous les chemins entre deux sommets (sans les cycles).

Exercice 7.2 - Triangles dans un graphe

Un triangle dans un graphe est un ensemble de trois sommets reliés entre eux par trois arêtes. La recherche et l'analyse des triangles dans un graphe sont importantes pour comprendre sa structure.

a. Déterminer mathématiquement le nombre de sous-ensembles de cardinal trois que possède un ensemble de sommets X.

b. Écrire une fonction retournant l'ensemble des triangles d'un graphe.

À chaque graphe $G = (X, E)$ correspond une unique matrice A symétrique de taille $n \times n$ avec $n = |X|$ définie par :

$$A_{ij} = \begin{cases} 1\,, & \text{si } \{i, j\} \in E\,, \\ 0\,, & \text{si } \{i, j\} \notin E\,. \end{cases}$$

Cette matrice est appelée la matrice d'adjacence du graphe G.

c. Définir une fonction retournant la matrice d'adjacence d'un graphe.

> **Indication**
>
> Faire attention que les sommets ne sont pas forcément indexés par des entiers compris entre 0 et n dans le dictionnaire.

d. Définir une fonction ayant pour argument une matrice d'adjacence et retournant le graphe correspondant sous forme d'un dictionnaire.

e. En utilisant la matrice d'adjacence A et la matrice $B = A^2$, écrire une fonction retournant l'ensemble des triangles d'un graphe.

f. En utilisant la matrice d'adjacence A, écrire une fonction calculant le nombre de triangles.

> **Indication**
>
> Interpréter les entrées de la matrice A^n.

Exercice 7.3 - Module NetworkX !!

De nombreux algorithmes de la théorie des graphes sont implémentés dans le module NetworkX, voir la documentation ici.

a. Suivre le tutoriel de NetworkX disponible ici.

b. Analyser un des graphes téléchargeables à l'adresse : https://github.com/gephi/gephi/wiki/Datasets ou https://snap.stanford.edu/data/.

8 Calcul symbolique

En tant que langage généraliste, Python n'inclut pas par défaut certains concepts mathématiques. Un exemple déjà vu concerne les vecteurs et les matrices numériques qui sont implémentés dans le module NumPy. Le but ici est d'introduire le module SymPy qui permet de faire du calcul symbolique.

Par exemple, le nombre $\sqrt{8}$ est représenté par défaut en Python comme un flottant. L'avantage de SymPy est que $\sqrt{8}$ est gardé en tant que racine et même automatiquement simplifié :

```
import sympy as sp
sp.sqrt(8)
```

$2\sqrt{2}$

À noter que la deuxième instruction n'est pas nécessaire, mais permet de présenter les résultats de manière plus élégante dans Jupyter Lab. La documentation de SymPy est disponible ici.

Concepts abordés

- symboles et expressions symboliques
- simplification
- analyse infinitésimale (limite, dérivation, intégration, développement limité)
- preuve assistée par ordinateur
- fonction pathologique
- fonction de Green
- coordonnées sphériques

Exercice 8.1 - Introduction à SymPy

Avant de pouvoir utiliser des variables symboliques, il faut les déclarer comme symboles :

```
x = sp.Symbol("x") # définit le symbole x
y = sp.Symbol("y", real=True) # définit la variable réelle y
e = sp.Symbol(r"\varepsilon", real=True, positive=True) # définit epsilon
 ↪  positif
```

Ensuite il est possible de faire des opérations entre symboles :

```
x + 2*y + e/4 + x**2 + 3*x + 2*y
```

$$\frac{\varepsilon}{4} + x^2 + 4x + 4y$$

La plupart des fonctions mathématiques sont implémentées symboliquement dans SymPy et il est également possible de les simplifier :

```
expr = sp.cos(x)**2 + sp.sin(x)**2 + (y**3 + y**2 - y - 1)/(y**2 + 2*y + 1)
 ↪  + sp.exp(-e)
sp.simplify(expr)
```

$$y + e^{-\varepsilon}$$

Finalement il est possible de faire des substitutions :

```
expr.subs(x,y) # remplace x par y
expr.subs({y:x, e:y}) # remplace y par x et e par y
```

$$\sin^2(x) + \cos^2(x) + e^{-x} + \frac{x^3 + x^2 - x - 1}{x^2 + 2x + 1}$$

puis par exemple de simplifier l'expression et de tracer son graphe en fonction de x comme sur la Figure 8.1 :

```
f = sp.simplify(expr.subs({y:x, e:y}))
sp.plot(f,(x,-2,6), title=f"Graphique de ${sp.latex(f)}$")
```

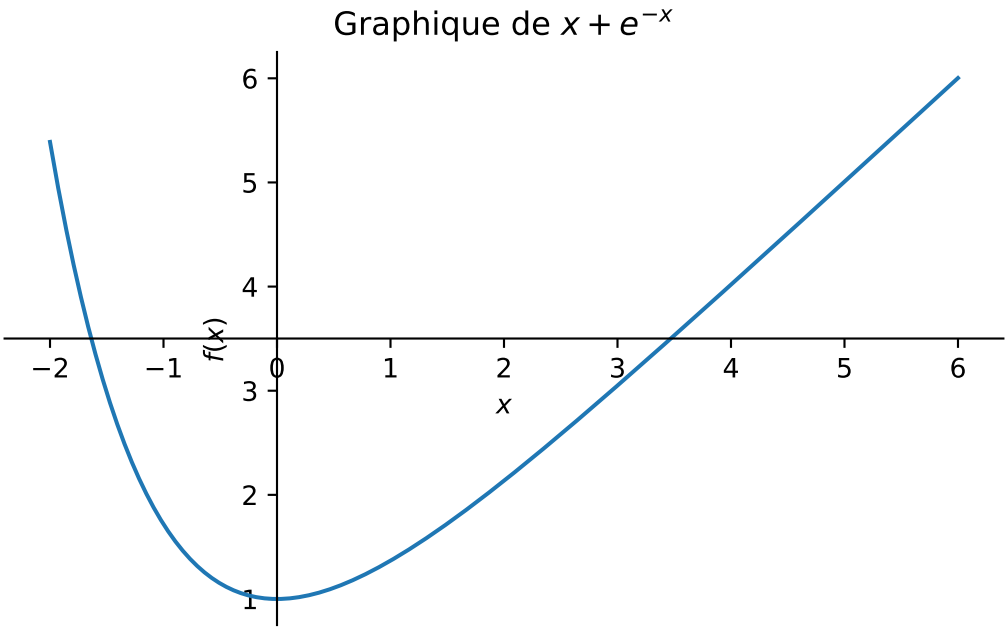


FIGURE 8.1 : Représentation graphique d'une fonction SymPy.

a. Lire la documentation de la fonction `solve` et l'utiliser pour calculer les racines d'un polynôme général de degré deux, puis de degré trois.

> **Indication**
>
> La documentation sur la résolution d'équations algébriques est disponible ici.

b. Lire la documentation des fonctions `evalf` et `N` pour évaluer l'expression $\frac{\pi^2}{4}$.

> **Indication**
>
> La documentation sur l'évaluation numérique est disponible ici.

c. Lire la documentation de la fonction `Rational` et évaluer numériquement le nombre rationel $\frac{43609}{999}$ avec 50 décimales.

d. Déterminer la partie réelle et imaginaire de l'expression :

$$\left(\frac{1+i\sqrt{3}}{1+i}\right)^{20}.$$

> **Indication**
>
> Voir la documentation ici.

e. Lire la documentation de la fonction `diff` et calculer la dérivée de $xe^{x^{x^x}}$ par rapport à x.

> **Indication**
>
> La documentation sur les dérivées est disponible ici.

f. Lire la documentation de la fonction `integrate` et calculer les intégrales suivantes :

$$I_1 = \int x^5 \sin(x)\,\mathrm{d}x\,, \qquad I_2 = \int_0^\infty \sin(x^2)\,\mathrm{d}x\,.$$

g. Calculer avec SymPy les limites suivantes :

$$L_1 = \lim_{x\to 0} \frac{\sin(x)}{x}\,, \qquad L_2 = \lim_{x\to 0} \sin\left(\frac{1}{x}\right), \qquad L_3 = \lim_{x\to\infty} \frac{5x^2+3x+2y}{y(x-4)(x-y)}\,.$$

h. Calculer le développement limité de $\tan(x)$ en $x = 0$ à l'ordre 10 et le développement asymptotique de $\left(1+\frac{1}{n}\right)^n$ pour $n \to \infty$ à l'ordre 5.

i. Déterminer les valeurs propres de la matrice :

$$\begin{pmatrix} 1 & a & 0 \\ a & 2 & a \\ 0 & a & 3 \end{pmatrix}.$$

Indication

La documentation sur les matrices symboliques est disponible ici.

Exercice 8.2 - Applications

Le but est d'utiliser SymPy pour résoudre symboliquement différents problèmes mathématiques en calculant le moins de choses possibles à la main.

a. Déterminer le nombre de zéros que contient l'entier 123!.

b. Déterminer le rapport entre la hauteur et le rayon d'un cylindre de manière à minimiser son aire à volume fixé.

c. Pour $x, y \in \mathbb{R}$ tels que $xy < 1$, démontrer que :

$$\arctan(x) + \arctan(y) = \arctan\left(\frac{x+y}{1-xy}\right).$$

Indication

Dériver l'équation par rapport à x et justifier.

d. Démontrer la formule suivante due à Gauss :

$$\frac{\pi}{4} = 12\arctan\left(\frac{1}{38}\right) + 20\arctan\left(\frac{1}{57}\right) + 7\arctan\left(\frac{1}{239}\right) + 24\arctan\left(\frac{1}{268}\right).$$

Il est impératif d'utiliser SymPy, la démonstration originale de Gauss faisant 25 pages, voir les pages 477 à 502 du deuxième volume de ses œuvres complètes disponible ici.

Indication

Appliquer la fonction tangente de chaque côté de l'équation puis simplifier. La documentation sur les différentes fonctions de simplifications est disponible ici.

e. Déterminer le volume de la région :

$$\left\{(x, y, z) \in \mathbb{R}^3 \,:\, x^2 + y^2 < z < 2x^2 + 4xy + 6y^2, |y| < 5, |x| < 4\right\}.$$

f. Déterminer l'expression des coefficients de Fourier réels de la fonction 2π-périodique f définie par $f(x) = |\sin(x)|$.

Exercice 8.3 - Conjecture due à Euler

Euler a conjecturé en 1769 qu'au moins k puissances k-ième d'entiers strictement positifs sont nécessaires pour que la somme soit elle-même une puissance k-ième. En d'autres termes si $n \geq 2$, $k \geq 1$, $a_1, a_2, \ldots, a_n \geq 1$ et $b \geq 1$ sont des entiers tels que :

$$\sum_{i=1}^{n} (a_i)^k = b^k$$

alors nécessairement $n \geq k$. Cette conjecture a été réfutée en 1966 par Lander & Parkin (doi:10.1090/S0002-9904-1966-11654-3) dans ce qui semble être le plus court article mathématique jamais écrit avec un contre-exemple pour $k = 5$:

$$27^5 + 84^5 + 110^5 + 133^5 = 144^5 .$$

Le but est de montrer que ce contre-exemple est le plus simple possible, dans le sens où il est l'unique contre-exemple avec $k \leq 5$ et $b \leq 144$.

a. Démontrer que la conjecture d'Euler est vraie pour $k = 1$ et $k = 2$.

b. Vérifier avec Python le contre-exemple ci-dessus.

c. Écrire une fonction `powers(bmax,k)` qui retourne l'ensemble (type `set`) de tous les entiers de 1 à `bmax` élevés à la puissance `k`.

d. Vérifier qu'il n'existe pas de contre-exemple avec $k = 3$ et $b \leq 144$.

e. Écrire une fonction `combinaisons(liste,n)` qui pour une liste d'entiers `liste` et un entier `n` retourne la liste de toutes les combinaisons de `n` entiers de `liste` classés par ordre croissant. Par exemple `combinaisons([1,2,3,4],2)` doit retourner `[(1,1), (1,2), (1,3), (1,4), (2,2), (2,3), (2,4), (3,3), (3,4), (4,4)]`.

> **Indication**
>
> Utiliser une fonction récursive sur `n`.

f. Écrire une fonction `test(bmax,n,k)` qui pour trois entiers `bmax`, `n` et `k` donnés, itère sur toutes les combinaisons de `n` entiers retournés par `combinaisons` et vérifie si la somme de ces `n` entiers élevés à la puissance `k` est un entier présent dans la liste `powers(bmax,k)`. Utiliser cette fonction pour vérifier qu'il n'existe pas de contre-exemple à la conjecture d'Euler pour $k = 4$ et $b \leq 144$. Suivant la puissance de votre ordinateur, il est possible de choisir également $k = 5$ et donc de vérifier que le contre-exemple de l'introduction est bien le plus simple.

Pour $k = 5$, la méthode précédente consistant à itérer sur toutes les combinaisons est plutôt lente. Une méthode plus rapide consiste à observer que l'ensemble des sommes du type :

$$(a_1)^5 + (a_2)^5 + (a_3)^5 + (a_4)^5 \,,$$

peut s'écrire comme $S_1 + S_2$ où S_1 et S_2 sont des sommes de deux entiers à la puissance cinq.

g. Écrire une fonction `sum2(bmax,k)` qui retourne un dictionnaire ayant pour clefs les sommes $(a_1)^k + (a_2)^k$ avec la valeur associée (a_1, a_2) pour $0 \leq a_1 \leq a_2 \leq$ `bmax`. On prendra soin d'enlever l'élément trivial zéro du dictionnaire. Dans la construction du dictionnaire, s'assurer qu'il est défini de manière unique dans le sens où il n'existe pas une autre valeur possible pour une clef existante. Tester `sum2(300,5)` et `sum2(300,3)`.

h. Utiliser le dictionnaire construit précédemment pour déterminer l'ensemble des contre-exemples pour $k = 5$ et $b \leq 300$ en itérant sur tous les éléments de `powers(bmax,5)` et de `sum2(bmax,5)`.

Indication

Une implémentation optimale prend au plus quelques secondes à s'exécuter.

Exercice 8.4 - Fonction pathologique

Le but est de construire une fonction qui visuellement semble régulière, mais qui en fait ne l'est pas. Soit la fonction $f : \mathbb{R} \to \mathbb{R}$ définie par :

$$f(x) = \sum_{k=1}^{\infty} \frac{\sin(k^2 x)}{k^5} \,.$$

Vu que la série converge absolument, la fonction f est bien définie.

a. À l'aide de SymPy calculer la fonction $g : \mathbb{R} \to \mathbb{R}$ définie en gardant les cent premiers termes de la série :

$$g(x) = \sum_{k=1}^{100} \frac{\sin(k^2 x)}{k^5} \,,$$

et représenter la fonction g graphiquement.

b. Estimer à la main l'erreur entre les fonctions f et g en valeur absolue.

c. Calculer la dérivée première et la dérivée seconde de g et représenter graphiquement ces deux dérivées. Que pouvez-vous conclure ?

d. Expliquer mathématiquement ce qui se passe.

Exercice 8.5 - Fonction de Green du laplacien !

Le but de cet exercice est de calculer entièrement automatiquement la fonction de Green du laplacien dans $\mathbb{R}^3$, *i.e.* la solution satisfaisant :

$$\Delta G(x) = \delta(x)\,,$$

dans $\mathbb{R}^3$ où $\delta(x)$ est la distribution de Dirac.

Pour cela, nous introduisons les coordonnées sphériques $x' = (r, \theta, \varphi)$ avec $r > 0$, $0 \leq \theta \leq \pi$ et $0 \leq \varphi < 2\pi$ caractérisées par :

$$\begin{aligned} x_1 &= r\cos\varphi\sin\theta \\ x_2 &= r\sin\varphi\sin\theta \\ x_3 &= r\cos\theta\,. \end{aligned}$$

a. Définir une fonction `to_spherical(expr)` permettant de convertir en coordonnées sphériques une expression donnée en coordonnées cartésiennes.

b. Définir une fonction `to_cartesian(expr)` permettant de convertir en coordonnées cartésiennes une expression donnée en coordonnées sphériques.

c. Calculer les facteurs d'échelle des coordonnées sphériques :

$$h_i = \left\| \frac{\partial x}{\partial x'_i} \right\|.$$

d. Définir une fonction `gradient(f)` permettant de calculer le gradient d'une fonction $f : \mathbb{R}^3 \to \mathbb{R}$ en coordonnées sphériques :

$$\nabla f = \left(\frac{1}{h_i} \frac{\partial f}{\partial x'_i} \right)_{i=1}^{3}.$$

e. Faire de même pour définir le laplacien en coordonnées sphériques :

$$\Delta f = \sum_{i=1}^{3} \frac{1}{J} \frac{\partial}{\partial x'_i} \left(\frac{J}{h_i^2} \frac{\partial f}{\partial x'_i} \right) \quad \text{où} \quad J = \prod_{i=1}^{3} h_i\,.$$

f. Trouver les solutions radiales (*i.e.* ne dépendant que de la variable r) de l'équation $\Delta G = 0$ dans $\mathbb{R}^3 \setminus \{0\}$.

Indication

Regarder la documentation de la fonction `dsolve` pour résoudre une équation différentielle.

g. Déterminer les équations que doivent satisfaire les constantes d'intégration pour que la solution précédente satisfasse en coordonnées cartésiennes :

$$\lim_{|x|\to\infty} G(x) = 0 \quad \text{et} \quad \Delta G(x) = \delta(x)\,.$$

Indication

Il faut transformer les deux conditions en coordonnées sphériques. La première condition s'exprime en coordonnées sphériques par :

$$\lim_{r\to\infty} G(r) = 0\,,$$

et la seconde est équivalente à :

$$\lim_{r\to 0} \int_0^{\pi}\int_0^{2\pi} \big(\nabla G(r)\cdot e_r\big) J(r,\theta,\varphi)\,\mathrm{d}\varphi\mathrm{d}\theta = 1\,.$$

h. Résoudre les équations sur les constantes d'intégrations et substituer dans la solution radiale de $\Delta G = 0$ pour obtenir l'expression de la fonction de Green du laplacien en coordonnées sphériques. Finalement déterminer la fonction de Green G du laplacien dans $\mathbb{R}^3$ en coordonnées cartésiennes.

i. !! Soit $g : \mathbb{R}^3 \to \mathbb{R}$ une fonction lisse à support compact invariante par rotations selon l'axe vertical. Déterminer le comportement asymptotique à grandes distances de la solution de l'équation :

$$\Delta f(x) = g(x)$$

jusqu'à l'ordre deux, *i.e.* les termes décroissant comme $|x|^{-1}$ et comme $|x|^{-2}$.

Indication

La solution est donnée par la formule de Green :

$$f(x) = \int_{\mathbb{R}^3} G(x-x_0)g(x_0)\,\mathrm{d}^3x_0 = \int_{B_R} G(x-x_0)g(x_0)\,\mathrm{d}^3x_0\,,$$

où R est choisi de sorte que le support de g soit contenu dans B_R. Pour calculer le développement asymptotique de cette intégrale, la première étape est de convertir $G(x-x_0)$ en coordonnées sphériques pour x et x_0. Vu que g est invariante par rotations selon l'axe vertical, alors en coordonnées sphériques g est indépendante de φ et est donnée par $g(r,\theta)$. La deuxième étape est de transformer l'intégrale en coordonnées sphériques avec une intégrale

triple sur r_0, θ_0 et φ_0. La troisième étape est de calculer le développement asymptotique de l'intégrant lorsque $r \to \infty$, jusqu'à l'ordre deux. Finalement, vu que r_0, θ_0 et φ_0 sont bornés, alors l'intégration commute avec le développement asymptotique et le résultat final est donné par l'intégration individuelle des deux termes du développement asymptotique.

9 Zéro de fonctions

Le but de cette série d'exercices est de déterminer les zéros de manière approchée d'une fonction notamment par la méthode de Newton. Cela permet en particulier de trouver des solutions approchées d'équations non linéaires. Cette méthode est fondamentale autant d'un point de vue numérique qu'analytique.

Concepts abordés

- méthode de Newton en une et plusieurs dimensions
- matrice jacobienne
- attracteur de la méthode de Newton
- ensemble fractal
- optimisation par parallélisation
- équation différentielle non linéaire
- différences finies

Exercice 9.1 - Méthode de Newton en une dimension

En une dimension, la méthode de Newton consiste à trouver une solution approchée d'une seule équation. Cette équation peut être mise sous la forme générale $F(x) = 0$ où $F : \mathbb{R} \to \mathbb{R}$ est une fonction assez régulière, donc le but est de trouver un zéro de la fonction F. L'équation $F(x) = 0$ est équivalente (si $F'(x) \neq 0$) à l'équation $G(x) = x$ où G est la fonction définie par :

$$G(x) = x - \frac{F(x)}{F'(x)}\,.$$

La méthode de Newton consiste à trouver un point fixe de G, *i.e.* résoudre $G(x) = x$ par itérations successives :

$$x_{i+1} = G(x_i) = x_i - \frac{F(x_i)}{F'(x_i)}$$

à partir d'une valeur initiale $x_0 \in \mathbb{R}$. Lorsque la suite $(x_i)_{i\in\mathbb{N}}$ converge, alors la limite x est une solution de $G(x) = x$ donc de $F(x) = 0$.

a. Écrire une fonction `newton1d(F, DF, x0, eps=1e-10, N=1000)` qui pour une fonction F, sa dérivée F' et une valeur initiale x_0 données calcule les itérations de Newton jusqu'à ce que

$|F(x_i)| < \varepsilon$ et retourne x_i. Si N itérations n'ont pas suffi à atteindre ce critère de convergence, alors retourner une erreur.

b. En utilisant la fonction définie précédemment, trouver une solution approchée de l'équation $e^{-x} = x$.

c. Sans utiliser les fonctions `sqrt` et `log` ni les puissances fractionnaires, définir une fonction `racine(x,n)` qui calcule $\sqrt[n]{x}$.

Parfois, la dérivée de la fonction F n'étant pas calculable analytiquement, il est alors nécessaire de l'approcher numériquement :

$$F'(x_i) \approx \frac{F(x_i) - F(x_{i-1})}{x_i - x_{i-1}},$$

ce qui mène à la méthode de la sécante :

$$x_{i+1} = x_i - F(x_i)\frac{x_i - x_{i-1}}{F(x_i) - F(x_{i-1})} = \frac{x_{i-1}F(x_i) - x_iF(x_{i-1})}{F(x_i) - F(x_{i-1})},$$

où x_0 et x_1 doivent être choisis.

d. Écrire une méthode `secant1d(F, x0, x1, eps=1e-10, N=1000)` implémentant la méthode de la sécante et la tester sur l'exemple précédent.

Exercice 9.2 - Méthode de Newton en plusieurs dimensions

La méthode de Newton en une dimension est facilement généralisable à plusieurs dimensions pour résoudre des équations de la forme $F(x) = 0$ où $F : \mathbb{R}^n \to \mathbb{R}^n$ est une fonction assez régulière. Conceptuellement la méthode est identique : l'équation $F(x) = 0$ est équivalente à $G(x) = x$ avec la fonction G définie par :

$$G(x) = x - \big(F'(x)\big)^{-1}F(x),$$

où $F'(x)$ désigne la matrice jacobienne de taille $n \times n$ de F en x. Ainsi les itérations de Newton s'écrivent :

$$x_{i+1} = x_i - \big(F'(x_i)\big)^{-1}F(x_i).$$

a. Écrire une fonction `newton(F, DF, x0, eps=1e-12, N=10000)` implémentant la méthode de Newton en plus d'une dimension.

Indication

Pour avoir une performance optimale, il ne faut pas inverser la matrice jacobienne mais résoudre un système linéaire avec $F(x_i)$ comme second membre.

b. Utiliser la fonction précédente pour résoudre le système suivant :

$$\cos(x) = \sin(y)\,, \qquad\qquad e^{-x} = \cos(y)\,.$$

Exercice 9.3 - Attracteur de la méthode de Newton

Le but de cet exercice est de résoudre l'équation $z^3 = 1$ dans le plan complexe à l'aide de la méthode de Newton et d'analyser vers laquelle des trois racines de l'unité la méthode va converger suivant le choix du point initial z_0.

a. Si nécessaire adapter la fonction `newton1d` pour que celle-ci s'applique également aux nombres complexes et la tester pour résoudre $z^3 = 1$ à partir de différentes valeurs de z_0.

b. Déterminer pour chaque $z_0 \in \{x_0 + iy_0 \,:\, x_0 \in [-3,3] \text{ et } y_0 \in [-3,3]\}$ vers quelle racine de l'unité la méthode de Newton va converger. Représenter graphiquement cet ensemble comme sur la Figure 9.1.

Indication

La fonction `meshgrid` de NumPy peut être utile pour construire la matrice correspondant à l'ensemble des z_0.

c. ! La méthode précédente a le désavantage de procéder séquentiellement au calcul pour chaque valeur de z_0, ce qui rend cette évaluation assez lente. Proposer une nouvelle implémentation permettant de calculer parallèlement toutes les valeurs de z_0 en utilisant les indexages NumPy.

Indication

Pour encore plus de rapidité, les itérations de Newton de $F(z) = z^3 - 1$ peuvent être calculées à la main :

$$z_{n+1} = \frac{1}{3z_n^2} + \frac{2z_n}{3}\,.$$

Exercice 9.4 - Équation différentielle non linéaire !!

Le but est de résoudre l'équation différentielle suivante avec conditions de valeurs limites :

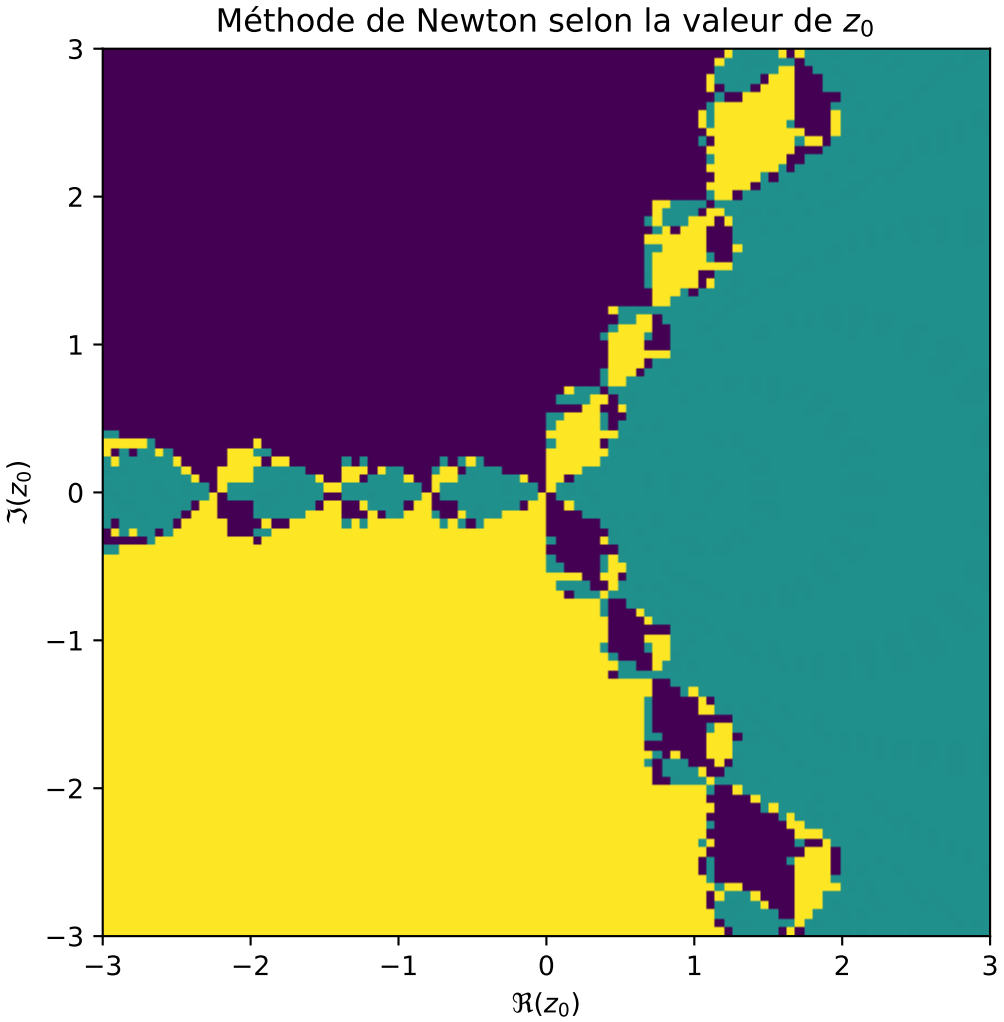


FIGURE 9.1 : Représentation graphique de la convergence de la méthode de Newton en fonction de la valeur z_0 de la donnée initiale.

$$u''(x) + u^3(x) = \sin(x)\,, \qquad u(0) = u(2\pi) = 0\,,$$

sur l'intervalle $[0, 2\pi]$. Cette équation est un modèle simplifié pour une équation de Schrödinger non linéaire.

La méthode employée est celle des différences finies qui consistent à chercher les valeurs de u aux points $x_n = \frac{2\pi n}{N}$ pour $n = 0, 1, \dots, N$. Les inconnues sont alors les nombres $u_n = u(x_n)$ et forment un vecteur de dimension $N+1$. La méthode des différences finies consiste à approximer la dérivée seconde par :

$$u''(x) \approx \frac{u(x+h) - 2u(x) + u(x-h)}{h^2}\,,$$

lorsque h est petit. En prenant $h = \frac{2\pi}{N}$, alors :

$$u''(x_n) \approx \frac{u_{n+1} - 2u_n + u_{n-1}}{h^2}\,,$$

et donc l'équation initiale s'approxime par :

$$\frac{u_{n+1} - 2u_n + u_{n-1}}{h^2} + u_n^3 = \sin(x_n)\,, \qquad u_0 = u_N = 0\,,$$

pour $n = 1, 2, \dots, N-1$. Cette équation peut être vue comme une équation du type $F(u) = 0$ pour $u = (u_n)_{n=0}^{N+1}$ et donc être résolue par la méthode de Newton.

a. Montrer l'approximation suivante :

$$u''(x) = \frac{u(x+h) - 2u(x) + u(x-h)}{h^2} + O(h^2) \quad \text{lorsque} \quad h \to 0\,.$$

Indication

Utiliser le théorème de Taylor.

b. Définir un vecteur `x` représentant les $N+1$ points équidistribués dans $[0, 2\pi]$ et `h` la distance entre les points, avec par exemple $N = 200$.

c. Définir une fonction `F(u)` représentant la fonction $F : \mathbb{R}^{N+1} \to \mathbb{R}^{N+1}$ permettant de mettre l'équation approchée sous la forme $F(u) = 0$.

Indication

Pour avoir une implémentation rapide, il est impératif pour construire F d'utiliser non pas une boucle, mais le slicing NumPy.

d. Définir une fonction `DF(u)` représentant la jacobienne de la fonction précédente.

Indication

La jacobienne est la dérivée de $F(u) = F(u_0, u_1, \dots, u_N)$ par rapport à $u = (u_0, u_1, \dots, u_N)$, c'est-à-dire :

$$F'(u) = \begin{pmatrix} \partial_0 F(u) & \partial_1 F(u) & \partial_2 F(u) & \cdots & \partial_{N-1} F(u) & \partial_N F(u) \end{pmatrix},$$

et peut se calculer explicitement à la main.

e. Utiliser la fonction `newton` définie précédemment pour calculer une solution approchée de l'équation. En changeant de valeurs initiales, est-il possible de trouver d'autres solutions ?

Indication

Essayer avec la donnée initiale $u_0(x) = (1 + k)\sin(kx)$ pour $k = 1, 2, 3, 4$ comme point de départ de la méthode de Newton.

10 Probabilités et statistiques

Dans un premier temps, la statistique de la proportion de nombres commençant par un certain chiffre sera étudiée. Puis dans un second, des modèles probabilistes importants seront introduits et simulés, comme les marches aléatoires, des illustrations du théorème central limite ou la percolation.

Concepts abordés

- statistiques et probabilités
- série harmonique aléatoire
- marche aléatoire
- théorème central limite
- vecteurs aléatoires
- percolation
- transition de phase
- histogrammes
- optimisation par compilation

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

Exercice 10.1 - Série harmonique de signe aléatoire

Le but de cet exercice est de simuler la convergence d'une série harmonique dont le signe est tiré aléatoirement. Plus précisément si $(X_i)_{i\in\mathbb{N}}$ est une suite de variables aléatoires indépendantes valant -1 ou 1 avec probabilité $\frac{1}{2}$, alors on définit la somme partielle :

$$W_0 = 0\,, \qquad W_n = \sum_{i=1}^{n} \frac{X_i}{i}\,,$$

et la question est de déterminer si la suite $(W_n)_{n\in\mathbb{N}}$ converge et si oui vers quoi.

a. Écrire une fonction `sign()` qui simule la variable aléatoire X_i.

b. Écrire une fonction `simulate(n)` qui retourne une liste avec une réalisation de $(W_0, W_1, W_2, \dots, W_n)$.

c. Faire la représentation graphique de la fonction $n \mapsto W_n$ pour différentes réalisations par exemple pour $0 \leq n \leq 1\,000$ et émettre une conjecture quant à la convergence de la suite $(W_n)_{n\in\mathbb{N}}$.

d. ! Déterminer l'histogramme de $W_{1\,000}$ pour 10^4 ou 10^5 réalisations pour avoir une idée de la loi de la variable aléatoire limite.

Exercice 10.2 - Ruine du joueur

Le but est de simuler l'évolution de la somme d'argent d'un joueur jouant à pile ou face. À chaque lancer le joueur gagne un euro si c'est pile et en perd un si c'est face. La probabilité d'obtenir pile est notée p, celle d'obtenir face q. En particulier $p = q = \frac{1}{2}$ si la pièce est équilibrée.

Mathématiquement, la somme S_i possédée par le joueur au temps i est donnée par une marche aléatoire :

$$S_i = \begin{cases} 0\,, & \text{si } S_{i-1} = 0\,, \\ S_{i-1} + X_i\,, & \text{si } S_{i-1} \geq 1\,, \end{cases}$$

où les $(X_i)_{i\geq 1}$ sont des variables aléatoires indépendantes de loi $\mathbb{P}(X_i = 1) = p$ et $\mathbb{P}(X_i = -1) = q$.

a. Écrire une fonction `simulate(p,k,N)` qui génère une réalisation de longueur N du processus à partir de $S_0 = k$, c'est-à-dire qui retourne $(S_0, S_1, S_2, \dots, S_N)$. Représenter graphiquement plusieurs réalisations.

b. Simuler un joueur qui, commençant avec une somme k, joue jusqu'à tout perdre ou avoir la somme $n \geq k$.

c. Si T désigne le temps auquel le jeu s'arrête, *i.e.* lorsque $S_T = 0$ ou $S_T = n$, retrouver par simulation les résultats théoriques sur le temps moyen :

$$\mathbb{E}(T) = \begin{cases} k(n-k)\,, & \text{si } p = q\,, \\ \dfrac{n}{p-q}\dfrac{1-\rho^k}{1-\rho^n} - \dfrac{k}{p-q}\,, & \text{si } p \neq q\,, \end{cases}$$

et le lieu de sortie :

$$\mathbb{P}(S_T = 0) = \begin{cases} \dfrac{n-k}{n}\,, & \text{si } p = q\,, \\ \dfrac{\rho^k - \rho^n}{1-\rho^n}\,, & \text{si } p \neq q\,, \end{cases}$$

où $\rho = q/p$. Pour cela on pourra faire un graphique de ces quantités en fonction de p ou se contenter de considérer le cas $p = q = \frac{1}{2}$.

Exercice 10.3 - Urnes de Polya

Une urne contient initialement (à $t = 0$) r_0 boules rouges et b_0 boules blanches. À chaque instant, on tire une boule uniformément au hasard dans l'urne. On remet ensuite cette boule dans l'urne et on y ajoute une boule de la même couleur. Un tel système s'appelle une *urne de Polya*. Le but de cet exercice est d'étudier le comportement de la fraction de boules rouges dans l'urne, c'est-à-dire le nombre de boules rouges sur le nombre total. On appellera respectivement r_n et b_n le nombre de boules rouges et blanches présentes dans l'urne à l'instant n.

a. Écrire une fonction `densite` prenant en argument un tuple représentant le nombre de boules rouges et blanches dans une urne, et qui renvoie la densité de boules rouges.

On veut construire de manière récursive la distribution du nombre de boules rouges au temps n, c'est-à-dire la liste des probabilités que le nombre de boules rouges soit égal à un entier donné k (qui sera l'indice de la liste). Cela se fait en écrivant deux fonctions : `next_dist_rouge`, prenant en argument la distribution à un instant n et qui renvoie celle à l'instant $n + 1$, qui est donc la fonction qui fait tout le travail, et `dist_rouge` qui est la fonction d'enrobage, prenant en argument r_0, b_0 et le temps n et qui nous renvoie la distribution au temps n par un appel récursif. On utilisera les faits utiles suivants (faire un petit dessin) :

- La distribution passée en argument à `next_dist_rouge` est une liste `r` et `r[k]` représente la probabilité d'avoir `k` boules rouges dans l'urne à l'instant n. Les indices pour `r` varient de 0 au nombre total `s` de boules à l'instant n.
- Au temps $n + 1$, pour avoir k boules rouges, il faut :
  - soit avoir eu k boules rouges à l'instant précédent et ne pas avoir tiré une boule rouge ;
  - soit avoir eu $k - 1$ boules rouges à l'instant précédent et avoir tiré une boule rouge.
- Si $n = 0$, le résultat de `dist_rouge` est complètement déterministe et les coefficients de la liste ne sont que des 0 et 1, dépendant de r_0 et b_0.

b. Écrire les fonctions `next_dist_rouge` et `dist_rouge` en utilisant les indications fournies. Regarder le résultat de `dist_rouge(0,1,n)` et `dist_rouge(1,1,n)` pour différentes valeurs de `n` (1, 2, 5, 10, 20, ...) et commenter.

Plutôt que de calculer théoriquement pour chaque n la suite des probabilités théoriques, nous allons faire des statistiques sur un grand nombre d'évolutions d'urnes de Polya, au bout d'un grand nombre d'étapes. Pour cela, il nous faut une fonction pour faire évoluer une urne de Polya.

c. Définir une fonction `polya_step(r,b)` qui, à partir de la composition d'une urne passée sous forme de deux paramètres `r` et `b`, renvoie l'évolution (aléatoire) après une étape de la composition de l'urne sous forme d'un tuple. Définir également une fonction `polya(r0,b0,N)` prenant en argument r_0, b_0 et N en paramètres et renvoyant la composition (aléatoire) d'une urne de Polya au bout de N étapes, aussi sous forme de tuple.

d. Écrire une fonction `data_rdens_polya(r0,b0,N,nbexp)` qui renvoie une liste de longueur `nbexp` contenant les densités de `nbexp` réalisations d'urnes de Polya au temps `N` initialisées avec `r0` boules rouges et `b0` boules blanches.

e. Stocker dans une variable le résultat de `data_rdens_polya(2,3,1000,10_000)` et dessiner un histogramme pour voir la répartition des densités. Attention, on veut que les hauteurs des barres soient normalisées pour que leur surface représente la proportion de points, et non pour qu'elles donnent le nombre de points par *bin*.

Indication

Une bonne règle de départ est de choisir le nombre de boîtes (*bins*) pour un histogramme de l'ordre de la racine carrée du nombre de points. Consulter la documentation de la fonction `hist` de Matplotlib.

Exercice 10.4 - Théorème central limite

Le théorème central limite (aussi appelé théorème limite central, théorème de la limite centrale ou théorème de la limite centrée) établit la convergence en loi de la somme d'une suite de variables aléatoires vers la loi normale. Intuitivement, ce résultat affirme qu'une somme de variables aléatoires identiques et indépendantes tend (sous certaines conditions) vers une variable aléatoire gaussienne. En voici une formulation :

Théorème : Soit (X_n) une suite de variables aléatoires réelles indépendantes et de même loi admettant une espérance μ et un écart-type $\sigma \neq 0$. Soit $(\bar{X}_n)$ la suite définie par

$$\bar{X}_n = \frac{1}{n}\sum_{k=1}^{n} X_k \,.$$

Pour n assez grand, la loi de $\bar{X}_n$ peut être approchée par la loi normale $\mathcal{N}(\mu, \frac{\sigma^2}{n})$.

L'objectif de cet exercice est de vérifier si ce théorème est valable pour différentes lois de probabilité :

— *Loi de Poisson* : distribution discrète sur $\mathbb{N}$, de paramètre λ, définie par :

$$\mathbb{P}(X = k) = \exp(-\lambda)\frac{\lambda^k}{k!}\,, \quad \forall k \in \mathbb{N}\,.$$

— *Loi normale* : distribution continue sur $\mathbb{R}$, de paramètres m et σ, définie par la densité :

$$\frac{1}{\sqrt{2\pi\sigma^2}}\exp\left(-\frac{(x-m)^2}{2\sigma^2}\right), \quad \forall x \in \mathbb{R}\,.$$

— *Loi de Cauchy* : distribution continue sur $\mathbb{R}$, de paramètres a et γ, définie par la densité :

$$\frac{1}{\pi\gamma\left(1+\left(\frac{x-a}{\gamma}\right)^2\right)}, \quad \forall x \in \mathbb{R}.$$

a. Définir une fonction `densite_loi_normale(x, mu var)` prenant comme un argument :

- `x` (tableau) : un tableau de nombres flottants
- `mu` (nombre flottant) : moyenne μ
- `var` (nombre flottant strictement positif) : variance σ^2

et qui renvoie la densité de la loi normale évaluée pour chaque nombre x dans `x` :

$$\mathcal{N}(\mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right).$$

b. Regarder la documentation de la fonction `numpy.random.poisson` et générer 10 valeurs aléatoires selon une loi de Poisson de paramètre $\lambda = 2$.

c. Écrire une fonction `echantillons_poisson(lam, N, M)` qui prend en arguments :

- `lam` (réel strictement positif) : paramètre λ pour la loi de Poisson
- `N` (entier strictement positif) : nombre d'expériences
- `M` (entier strictement positif) : nombre de variables aléatoires générées pour chaque expérience

et qui génère `N` expériences avec `M` variables aléatoires générées par expérience, et qui renvoie :

- la valeur moyenne (nombre flottant) sur les `N * M` variables aléatoires générées
- l'écart-type (nombre flottant) sur les `N * M` variables aléatoires générées
- un vecteur numpy de taille `N`, où chaque élément est la moyenne des variables aléatoires d'une expérience.

Pour les valeurs `lam=2` et `N=10_000`, exécuter `echantillons_poisson(lam, N, M)` pour $M \in \{10, 100, 1\,000\}$ et sauvegarder les résultats dans des variables.

d. Pour chaque valeur de `M`, afficher la distribution du vecteur numpy contenant les moyennes de chaque expérience, ainsi que la distribution de la loi normale attendue si la théorème central limite est vérifié. On pourra utiliser la fonction `hist` de Matplotlib pour afficher l'histogramme d'un tableau, avec le paramètre `bins=50` pour fixer le nombre colonnes et `density=True` pour afficher une distribution de probabilité. On utilisera les moyennes et les écart-types empiriques renvoyés par la fonction `echantillons_poisson` pour les paramètres de la loi normale. Choisir des valeurs pertinentes pour les valeurs limites de l'axe des abscisses. Faire une hypothèse sur la validité du théorème central limite pour la loi de Poisson.

e. Reprendre les questions *c)* et *d)* pour la loi normale. Pour cela, utiliser la fonction `numpy.random.normal` puis générer des moyennes avec les paramètres `loc=2`, `scale=1`, `N=10_000` et $M \in \{10, 100, 1\,000\}$. Tracer les histogrammes et faire une hypothèse sur la validité du théorème central limite pour la loi normale.

f. Reprendre les questions *c)* et *d)* pour la loi de Cauchy avec $a = 0$ et $\gamma = 1$. Pour cela utiliser la fonction `numpy.random.standard_cauchy`. Utiliser `bins=np.arange(-10, 10.1, 0.1)` et faire une hypothèse sur la validité du théorème central limite pour la loi de Cauchy.

Exercice 10.5 - Génération aléatoire de vecteurs unitaires

L'objectif de cet exercice est de trouver une méthode efficace pour générer aléatoirement des vecteurs unitaires dans $\mathbb{R}^n$ selon une loi uniforme. Nous allons commencer par le cas $n = 2$, où l'on peut représenter un vecteur réel par un nombre complexe.

a. On considère la stratégie suivante pour générer aléatoirement un vecteur unitaire dans $\mathbb{R}^2$:

1. Générer x aléatoirement selon la loi uniforme sur $[-1, 1]$.
1. Générer y aléatoirement selon la loi uniforme sur $[-1, 1]$.
1. Renvoyer le complexe unitaire $z = \frac{x}{\sqrt{x^2+y^2}} + \frac{y}{\sqrt{x^2+y^2}}i$.

Écrire une fonction `generer_complexe` qui prend en argument un entier positif N et qui renvoie un tableau NumPy de taille N, où chaque élément est un complexe généré par la stratégie ci-dessus.

> **Indication**
>
> Il est beaucoup plus efficace de générer directement N variables aléatoires en utilisant l'argument `size` de la fonction `numpy.random.uniform` que de les générer une par une avec une boucle `for`.

b. Pour $n = 2$, une manière de vérifier si la distribution des vecteurs est uniforme est de regarder la distribution des angles/arguments des nombres complexes (c'est-à-dire $\arg z$) : celle-ci doit être uniforme. Utiliser la fonction `generer_complexe` avec $N = 10^6$ et afficher la distribution des angles/arguments. La stratégie présentée à la question précédente permet-elle de générer des vecteurs unitaires de manière uniforme ?

> **Indication**
>
> On pourra utiliser la fonction `numpy.angle`.

c. On propose cette modification à la stratégie précédente :

1. Générer x aléatoirement selon la loi uniforme sur $[-1, 1]$.
1. Générer y aléatoirement selon la loi uniforme sur $[-1, 1]$.
1. Si $x^2+y^2 \leq 1$, renvoyer le complexe unitaire $z = \frac{x}{\sqrt{x^2+y^2}} + \frac{y}{\sqrt{x^2+y^2}}i$ (sinon ne rien renvoyer du tout).

Écrire une fonction `generer_complexe_monte_carlo` qui prend en argument un entier positif N, correspondant au nombre de vecteurs candidats et qui renvoie un tableau NumPy de taille $n \leq N$, où chaque élément est un complexe généré par la stratégie ci-dessus. *Attention : n* est aléatoire et ne peut donc pas être déterminé à l'avance.

Reprendre la question précédente et afficher la distribution des angles/arguments des complexes générés par cette stratégie. Cette stratégie permet-elle de générer des vecteurs unitaires de manière uniforme ?

d. On se demande maintenant combien de complexes candidats il faut générer en moyenne pour en accepter n, ce qui est équivalent à déterminer combien de complexes sont en moyenne acceptés parmi les N candidats. La loi des grands nombres permet d'affirmer que le ratio converge vers la probabilité qu'un vecteur candidat soit accepté. Cette probabilité est égale au ratio de la surface d'inclusion π (le cercle unitaire) sur la surface totale 4 (le carré $[-1, 1]^2$), c'est-à-dire $\frac{\pi}{4}$. Comparer le ratio $\frac{n}{N}$ à $\frac{\pi}{4}$.

e. On se place maintenant dans le cas général $n \geq 2$. La stratégie considérée est la même que celle à la question *c)* :

1. Générer un vecteur $x = (x_1, \dots, x_n)$ où chaque x_k est généré aléatoirement et de manière indépendante selon la loi uniforme sur $[-1, 1]$.

1. Si $\|x\|_2 \leq 1$, renvoyer le vecteur unitaire $\frac{x}{\|x\|_2}$.

Le volume de la boule unitaire dans $\mathbb{R}^n$ est donné par :

$$V_n = \frac{\pi^{n/2}}{\Gamma(\frac{n}{2} + 1)}$$

et le volume du cube $[-1, 1]^n$ est 2^n. Afficher sur un graphique la probabilité qu'un vecteur soit accepté à l'étape 2 de cette stratégie pour $n \in \{2, \dots, 20\}$. Pensez-vous que cette stratégie est efficace pour de grandes valeurs de n ?

Indication

On pourra utiliser `scipy.special.gamma` pour la fonction Γ et la fonction `scatter` de Matplotlib pour afficher graphiquement les valeurs d'une suite avec une échelle adaptée.

f. Il est possible de montrer que la stratégie suivante permet de générer des vecteurs aléatoires sur $\mathbb{R}^n$:

1. Générer un vecteur $x = (x_1, \dots, x_n)$ où chaque x_k est généré aléatoirement et de manière indépendante selon la loi normale centrée réduite : $x_k \sim \mathcal{N}(0, 1)$.
2. Renvoyer le vecteur unitaire $\frac{x}{\|x\|_2}$.

Vérifier pour $n = 2$ que cette stratégie génère bien aléatoirement des vecteurs unitaires selon une loi uniforme en affichant la distribution des angles (en représentant les vecteurs comme des nombres complexes) pour $N = 10^6$ vecteurs.

Indication

On pourra utiliser la fonction `numpy.random.standard_normal`.

Exercice 10.6 - Percolation !!

Le but est d'étudier un modèle de percolation dans un milieu poreux. Le milieu est modélisé par une matrice aléatoire de booléens qui détermine les sites qui peuvent être envahis par l'eau et ceux qui sont imperméables. Une matrice percole s'il existe un chemin d'eau allant de la ligne supérieure vers la ligne inférieure. Dans les exemples de la Figure 10.1, les entrées d'une matrice pouvant être envahies par l'eau sont colorées et les entrées effectivement remplies d'eau sont en bleu. La première matrice ne percole pas alors que la seconde oui.

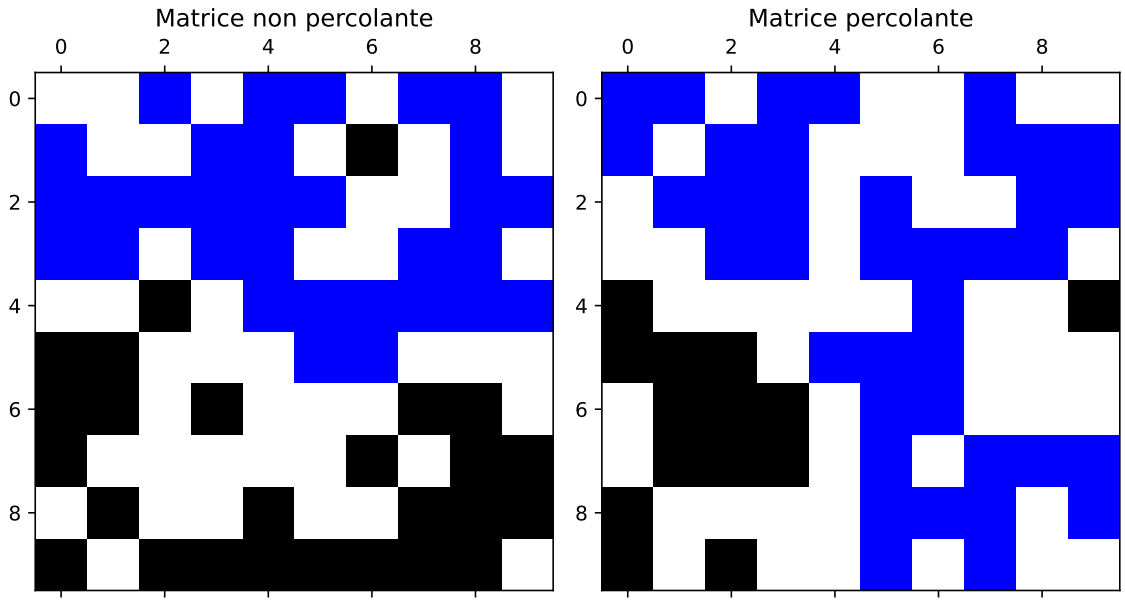


FIGURE 10.1 : La matrice de gauche ne percole pas alors que celle de droite oui.

a. Écrire une fonction `generate(n,p)` qui génère une matrice de booléens de taille $n \times n$ telle que chaque entrée ait une probabilité p d'être juste et $1 - p$ d'être fausse.

Indication

La fonction `random.binomial` de NumPy peut être utile.

b. Définir une fonction `fill(isopen)` qui pour une matrice de booléens donnée renvoie une autre matrice de booléens avec les entrées envahies par l'eau.

> **Indication**
>
> Définir une matrice de booléens `isfull` pour stocker si une entrée est remplie par l'eau ou pas, puis définir une fonction récursive `flow(isopen, isfull, i, j)` permettant d'envahir toutes les entrées possibles à partir de (i, j).

c. À l'aide de Matplotlib représenter le remplissage de différentes matrices générées aléatoirement.

d. Définir une fonction `percolate(isopen)` permettant de déterminer si une matrice de booléens percole ou non.

e. !! Calculer le temps nécessaire pour déterminer si une matrice de taille 50×50 avec $p = 0.9$ percole ou non. Lire la documentation du module Numba pour réduire le temps de calcul en compilant une des fonctions : https://numba.pydata.org/.

> **Indication**
>
> La fonction qui est la plus utilisée est la fonction récursive, donc c'est celle qu'il faut optimiser en la compilant.

f. En faisant des statistiques, déterminer la probabilité qu'une matrice aléatoire booléenne de taille $n \times n$ avec probabilité p percole. Étudier cette probabilité en fonction de p et de n.

> **Indication**
>
> Faire le graphique de cette probabilité de percolation en fonction de p pour différentes valeurs de n.

g. !!! Les statistiques effectuées au point précédent sont un exemple typique de calculs pouvant être facilement exécutés en parallèle, car chaque cas est indépendant des autres. Paralléliser l'algorithme précédent de manière à utiliser tous les cœurs de son processeur, par exemple à l'aide du module `mpi4py`.

> **Indication**
>
> L'utilisation de Jupyter Lab pour faire du calcul parallèle est assez complexe à mettre en œuvre, il vaut mieux utiliser la ligne de commande pour exécuter un script en parallèle, par exemple pour quatre cœurs : `mpirun -n 4 script.py`. À noter que Open MPI ou MPICH doit être installé sur l'ordinateur.

11 Équations différentielles

Le but est d'introduire les méthodes de base permettant de résoudre des équations différentielles ordinaires du premier ordre du type :

$$\dot{x}(t) = f(t, x(t)), \qquad x(0) = x_0,$$

où $f : \mathbb{R}^+ \times \mathbb{R}^n \to \mathbb{R}^n$ est une fonction assez régulière et $x_0 \in \mathbb{R}^n$ une donnée initiale. À noter que les équations différentielles ordinaires d'ordre supérieur peuvent être mises sous la forme précédente du premier ordre.

Concepts abordés

- méthodes d'Euler
- méthodes de Runge-Kutta
- équation aux dérivées partielles non linéaire
- différences finies
- méthodes adaptatives

Exercice 11.1 - Méthodes d'Euler

L'idée la plus simple pour résoudre de manière approchée une équation différentielle ordinaire est de discrétiser le temps avec un pas h et d'approximer la dérivée temporelle sur chaque intervalle de longueur h. Il y a deux façons simples d'approximer la dérivée en temps. La première est l'approximation par différence finie avant :

$$\dot{x}(t) \approx \frac{x(t+h) - x(t)}{h},$$

la seconde par différence finie arrière :

$$\dot{x}(t) \approx \frac{x(t) - x(t-h)}{h}.$$

Les inconnues étant les évaluations de la solution x aux temps $t_i = ih$ pour $i \geq 0$, c'est-à-dire $x_i = x(t_i)$. L'équation différentielle peut ainsi être approchée à l'aide des différences finies avant par :

$$\frac{x_{i+1} - x_i}{t_{i+1} - t_i} = f(t_i, x_i)\,,$$

ce qui donne la formule d'Euler explicite :

$$x_{i+1} = x_i + (t_{i+1} - t_i) f(t_i, x_i)\,.$$

Avec l'approximation par différences finies arrière, on obtient la méthode d'Euler implicite :

$$x_i = x_{i-1} + (t_i - t_{i-1}) f(t_i, x_i)\,.$$

La formule d'Euler explicite permet de calculer directement tous les x_i par récurrence en connaissant x_0. En revanche la formule d'Euler implicite nécessite à chaque pas de temps la résolution d'une équation non linéaire pour x_i par exemple avec la méthode de Newton.

a. Écrire une fonction `euler_explicit(f,x0,t)` qui pour une donnée initiale `x0` retourne les valeurs $x_0, x_1, \dots, x_m$ calculées avec la méthode d'Euler explicite aux temps $(t_i)_{i=0}^m$ représentés par le vecteur `t`.

b. Utiliser la méthode d'Euler explicite pour résoudre l'équation différentielle :

$$\dot{x}(t) + x(t) = \sin(t)\,, \quad x(0) = 1\,,$$

pour $t \in [0, 10]$. Comparer les résultats avec la solution exacte :

$$x(t) = \frac{1}{2}\big(\sin(t) - \cos(t) + 3e^{-t}\big)\,,$$

pour différentes discrétisations du temps.

c. Résoudre le problème précédent avec la méthode d'Euler implicite.

> **Indication**
>
> Vu que l'équation précédente est linéaire, on peut en fait rendre explicite la méthode d'Euler implicite en résolvant l'équation implicite à la main.

d. !! Définir une fonction `euler_implicit(f, Dxf, x0, t)` implémentant la méthode d'Euler implicite pour des équations non linéaires. À noter que pour résoudre le problème non linéaire avec la méthode de Newton, la dérivée de f selon x est nécessaire.

> **Indication**
>
> Il est également possible d'utiliser l'algorithme de recherche de zéro de fonction `optimize.fsolve` de Scipy qui ne nécessite pas de connaître la dérivée de f.

e. ! Utiliser les méthodes précédentes pour trouver une solution approchée du système :

$$\begin{aligned} \dot{x}(t) + \cos(y(t)) &= \sin(t)\,, & x(0) &= 1\,, \\ \dot{y}(t) + \cos(x(t)) &= 0\,, & y(0) &= 0\,. \end{aligned}$$

Exercice 11.2 - Méthodes de Runge-Kutta

Le but de cet exercice est d'introduire une classe de méthodes plus précises que les méthodes d'Euler pour résoudre des équations différentielles ordinaires. Au lieu de faire une approximation au premier ordre en h l'idée est de faire une approximation d'un ordre supérieur.

L'idée de base est de construire une suite x_i donnant une approximation de la solution de $\dot{x}(t) = f(t,x)$ au temps t_i pour $i \in \mathbb{N}$. Cette suite est définie par :

$$x_{i+1} = x_i + M(t_i, x_i, t_{i+1} - t_i)\,,$$

pour une certaine fonction M appelée méthode. Par exemple pour la méthode d'Euler explicite, la fonction M est donnée par :

$$M(t,x,h) = hf(t,x)\,.$$

Une méthode de Runge-Kutta d'ordre deux est donnée par :

$$M(t,x,h) = hf\left(t + \frac{h}{2}, x + \frac{h}{2} f(t,x)\right).$$

Une méthode de Runge-Kutta d'ordre quatre est donnée par :

$$M(t,x,h) = \frac{h}{6}\big(k_1 + 2k_2 + 2k_3 + k_4\big)\,,$$

où

$$\begin{aligned} k_1 &= f(t,x)\,, \\ k_2 &= f\left(t + \frac{h}{2}, x + \frac{h}{2}k_1\right), \\ k_3 &= f\left(t + \frac{h}{2}, x + \frac{h}{2}k_2\right), \\ k_4 &= f(t+h, x + hk_3)\,. \end{aligned}$$

À noter que plus généralement, une méthode de Runge-Kutta d'ordre s est donnée par :

$$M(t,x,h) = h\sum_{i=1}^{s} b_i k_i\,,$$

où

$$\begin{aligned}
k_1 &= f(t,x)\,,\\
k_2 &= f(t+c_2h, x+ha_{21}k_1)\,,\\
k_3 &= f(t+c_3h, x+h(a_{31}k_1+a_{32}k_2))\,,\\
&\vdots\\
k_s &= f(t+c_sh, x+h(a_{s1}k_1+a_{s2}k_2+\cdots+a_{s,s-1}k_{s-1}))\,.
\end{aligned}$$

Les coefficients a_{ij} (pour $1 \le j < i \le s$), c_i (pour $2 \le i \le s$) et b_i (pour $1 \le i \le s$), sont souvent représentés dans un tableau de Butcher :

$$\begin{array}{c|ccccc}
0 & & & & & \\
c_2 & a_{21} & & & & \\
c_3 & a_{31} & a_{32} & & & \\
\vdots & \vdots & & \ddots & & \\
c_s & a_{s1} & a_{s2} & \cdots & a_{s,s-1} & \\
\hline
 & b_1 & b_2 & \cdots & b_{s-1} & b_s
\end{array}$$

Par exemple, le tableau de Butcher de la méthode précédente d'ordre deux est :

$$\begin{array}{c|cc}
0 & & \\
\frac{1}{2} & \frac{1}{2} & \\
\hline
 & 0 & 1
\end{array}$$

et celui de la méthode d'ordre quatre :

$$\begin{array}{c|cccc}
0 & & & & \\
\frac{1}{2} & \frac{1}{2} & & & \\
\frac{1}{2} & 0 & \frac{1}{2} & & \\
1 & 0 & 0 & 1 & \\
\hline
 & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6}
\end{array}$$

a. Définir une fonction `integrate(f, x0, t, M)` qui pour une liste de temps $(t_i)_{i=0}^N$ donnée retourne les valeurs correspondantes $x_0, x_1, \dots, x_N$ avec la méthode M.

b. Implémenter les fonctions `M(f,t,x,h)` pour la méthode d'Euler explicite et la méthode de Runge-Kutta d'ordre deux. Comparer les deux méthodes.

c. Implémenter la fonction `M(f,t,x,h)` pour la méthode de Runge-Kutta d'ordre quatre. Comparer avec la méthode d'ordre deux.

Exercice 11.3 - Mouvement d'une planète

Le but est de simuler le mouvement bidimensionnel d'une planète gravitant autour d'une étoile fixe. L'étoile est supposée fixée à l'origine et la position de la planète dans le plan est décrite par le vecteur $x \in \mathbb{R}^2$. L'étoile est supposée interagir avec la planète avec le potentiel :

$$V(x) = \frac{1}{\alpha}|x|^\alpha ,$$

pour un certain $\alpha \in \mathbb{R}$ où $|x|$ désigne la norme euclidienne du vecteur x. À noter que le potentiel gravitationnel correspond à $\alpha = -1$. L'équation de la planète dans ce champ de force est donnée par :

$$\ddot{x} = -\nabla V(x) = -x|x|^{\alpha-2} .$$

a. Réécrire l'équation différentielle d'ordre deux comme une équation différentielle d'ordre un pour x et $p = \dot{x}$.

b. Implémenter la fonction `f(t,xp)` correspondant à l'équation trouvée au point précédent.

c. À l'aide de la méthode de Runge-Kutta d'ordre quatre, résoudre l'équation différentielle pour différentes données initiales et différentes valeurs de α et tracer les trajectoires $x(t)$ dans le plan. Interpréter les résultats et expliquer en particulier pourquoi les cas $\alpha = -1$ et $\alpha = 2$ sont différents des autres.

Exercice 11.4 - Attracteur de Lorenz

Le modèle de Lorenz est un système de trois équations différentielles couplées de la forme :

$$\begin{aligned}
\dot{x} &= \sigma(y - x) , \\
\dot{y} &= x(\rho - z) - y , \\
\dot{z} &= xy - \beta z ,
\end{aligned}$$

où ρ, σ, β sont trois paramètres réels. Il s'agit d'un modèle très simplifié de couplage entre l'atmosphère et l'océan proposé en 1963 par Edward Lorenz.

a. Écrire mathématiquement l'expression de la fonction $f : \mathbb{R} \times \mathbb{R}^3 \to \mathbb{R}^3$ permettant de mettre le système de Lorenz sous la forme :

$$\dot{x} = f(t, x)\,,$$

où x est le vecteur (x, y, z). Implémenter une fonction `f(t,x,rho,sigma,beta)` correspondant à la fonction f.

b. Écrire une fonction `plot_lorenz(rho,sigma,beta)` qui pour les paramètres ρ, σ, β donnés, représente graphiquement la trajectoire $(x(t), z(t))$ pour $t \in [0, 20]$ à partir de la donnée initiale $x_0 = (x_0, y_0, z_0) = (1, 1, 1)$. Utiliser par exemple la méthode de Runge-Kutta d'ordre quatre avec un pas de temps $\Delta t = 0.001$. Tester avec $\sigma = 10$, $\beta = 8/3$ et les valeurs $\rho = 10, 15, 20, 25$ et décrire ce qui est observé.

c. À l'aide de SymPy, déterminer les solutions stationnaires en fonction des paramètres ρ, σ, β, c'est-à-dire les solutions de $f(t, x) = 0$ pour tout $t > 0$. Interpréter les graphiques précédents au vu de cela.

Exercice 11.5 - Équation des ondes cubique !!

Le but est de résoudre numériquement l'équation des ondes non linéaire sur $\mathbb{R}$:

$$-\frac{\partial^2 u}{\partial t^2} + \frac{\partial^2 u}{\partial x^2} = u^3\,, \qquad u(0, \cdot) = u_0\,, \qquad \frac{\partial u}{\partial t}(0, \cdot) = v_0\,,$$

pour $u : \mathbb{R}^+ \times \mathbb{R} \to \mathbb{R}$ avec $u_0, v_0 : \mathbb{R} \to \mathbb{R}$ deux fonctions données.

> **Note**
>
> Les propriétés de cette équation d'apparence simple sont très mal comprises mathématiquement, voir l'article de recherche suivant pour plus de détails : doi:10.2140/apde.2012.5.411.

a. Réécrire l'équation précédente sous la forme de deux équations du premier ordre en temps pour u et $v = \frac{\partial u}{\partial t}$.

b. En approximant la seconde dérivée en espace par différences finies comme à l'Exercice 9.4 montrer que l'équation peut s'approximer de la manière suivante :

$$\begin{aligned} \frac{\partial u_n}{\partial t} &= v_n\,, & u_n(0) &= u_0(x_n)\,, \\ \frac{\partial v_n}{\partial t} &= \frac{u_{n-1} - 2u_n + u_{n+1}}{h^2} - u_n^3\,, & v_n(0) &= v_0(x_n)\,, \end{aligned}$$

où $(x_n)_{n=0}^N$ représente $N+1$ points équidistants de h dans l'intervalle $[-L, L]$ et $u_n(t) = u(t, x_n)$ et $v_n(t) = v(t, x_n)$. Pour les conditions au bord du domaine, *i.e.* lorsque $n = 0$ ou $n = N$, on prendra :

$$\frac{\partial v_0}{\partial t} = 0\,, \qquad \frac{\partial v_N}{\partial t} = 0\,.$$

c. Déterminer la fonction $f : \mathbb{R} \times \mathbb{R}^{2N+2} \to \mathbb{R}^{2N+2}$ permettant de mettre l'approximation précédente sous la forme $\dot{u} = f(t, u)$ pour $u = (u, v)$ et implémenter cette fonction.

d. Résoudre l'équation différentielle donnée par $\dot{u} = f(t, u)$ par exemple avec la méthode de Runge-Kutta d'ordre quatre. Un bon choix de paramètres est $L = 100$, $N = 1\,000$ et pour la donnée initiale $u_0(x) = e^{-x^2}$ et $v_0(x) = 0$. La vitesse de propagation de l'onde est un et donc, après un temps plus grand que L, l'onde sort de la boîte $[-L, L]$ et ne correspond plus à une bonne approximation de l'équation initiale.

e. À l'aide du module `animation` de Matplotlib réaliser une vidéo montrant l'évolution de l'onde en fonction du temps.

> **Indication**
>
> Utiliser par exemple la fonction `FFMpegWriter`.

Exercice 11.6 - Méthodes de Bogacki-Shampine !!!

En combinant deux méthodes de Runge-Kutta d'ordres différents (par exemple (2,3) ou (4,5)), on obtiendra une estimation empirique de l'erreur sur un pas de temps. En utilisant cette estimation d'erreur, il est possible d'adapter le pas de temps, soit en l'augmentant soit en le diminuant et ainsi de s'adapter à l'équation.

Pour une méthode de Runge-Kutta d'ordre s, une méthode interne d'ordre moins élevé (généralement $s-1$) est donnée par :

$$M^*(t, x, h) = h \sum_{i=1}^{s} b_i^* k_i\,,$$

où les k_i sont identiques à ceux de la méthode d'ordre s. Une estimation de l'erreur est alors donnée par :

$$E(t, x, h) = M(t, x, h) - M^*(t, x, h) = h \sum_{i=1}^{s} (b_i - b_i^*) k_i\,.$$

Une telle méthode est donnée par un tableau de Butcher étendu :

$$
\begin{array}{c|ccccc}
0 & & & & & \\
c_2 & a_{21} & & & & \\
c_3 & a_{31} & a_{32} & & & \\
\vdots & \vdots & & \ddots & & \\
c_s & a_{s1} & a_{s2} & \cdots & a_{s,s-1} & \\
\hline
 & b_1 & b_2 & \cdots & b_{s-1} & b_s \\
 & b_1^* & b_2^* & \cdots & b_{s-1}^* & b_s^*
\end{array}
$$

a. Implémenter la méthode de Bogacki-Shampine d'ordre (4,5). L'article original est disponible à l'adresse doi:10.1016/0898-1221(96)00141-1.

Indication

Les coefficients des tables de Butcher sont implémentés dans un module `nodepy` donc la documentation est disponible ici. Le nom de la méthode dans ce package est « BS5 ».

12 Science des données

La méthodologie de la science des données consiste à utiliser les données disponibles (généralement une grande quantité) pour répondre à des questions. Le traitement d'un grand nombre de données est peu compatible avec les structures de données par défaut de Python ou NumPy. Dans un premier temps, le module Pandas, spécialement conçu pour analyser les données, sera présenté. La documentation de Pandas est disponible ici.

Pour charger le module Pandas, il est habituel de procéder comme suit :

```
import pandas as pd
```

Ensuite, des données réelles seront analysées en faisant des statistiques sur la proportion de nombres commençant par un certain chiffre, ainsi qu'en déterminant des tendances. Finalement, trois aspects d'apprentissage automatiques (*machine learning*) seront vus, avec la reconnaissance de chiffres manuscrits, la différentiation automatique et l'utilisation d'un réseau de neurones.

Concepts abordés

- importation et analyse de données
- utilisation de Pandas
- loi de Benford
- méthodes des moindres carrés
- classification d'images
- différentiation automatique
- réseau de neurones

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

Exercice 12.1 - Introduction à Pandas

Création. Dans Pandas, une table de données est un `DataFrame` et la construction d'une table peut se faire manuellement à partir d'un dictionnaire, par exemple :

```
df = pd.DataFrame(
    {
        "Nom": [
            "M. Harris Braund",
            "M. William Allen",
```

```
            "Mme Elizabeth Bonnell",
        ],
        "Age": [22, 35, 58],
        "Genre": ["homme", "homme", "femme"],
    }
)
```

Dans Jupyter Lab, il suffit d'exécuter la cellule :

```
df
```

	Nom	Age	Genre
0	M. Harris Braund	22	homme
1	M. William Allen	35	homme
2	Mme Elizabeth Bonnell	58	femme

pour afficher le tableau. On peut voir ici qu'il se compose de trois colonnes (avec les étiquettes données) et de trois lignes étiquetées par défaut par des nombres entiers.

Extraction de colonnes. Une seule colonne d'un `DataFrame` est appelée une `Series` et peut être facilement extraite :

```
df["Age"]
```

```
0    22
1    35
2    58
Name: Age, dtype: int64
```

Sur une telle colonne, des statistiques peuvent être réalisées de manière simple, par exemple pour déterminer les personnes les plus âgées et l'âge moyen :

```
df["Age"].max(), df["Age"].mean()
```

```
(np.int64(58), np.float64(38.333333333333336))
```

Deux colonnes peuvent également être extraites :

```
df[["Age", "Genre"]]
```

	Age	Genre
0	22	homme
1	35	homme
2	58	femme

Création d'une nouvelle colonne. Il est également possible d'ajouter des données supplémentaires, par exemple une nouvelle colonne avec 100 divisé par les âges :

```
df["Age inv"] = 100/df["Age"]
df
```

	Nom	Age	Genre	Age inv
0	M. Harris Braund	22	homme	4.545455
1	M. William Allen	35	homme	2.857143
2	Mme Elizabeth Bonnell	58	femme	1.724138

Ces opérations se font élément par élément. Il n'est donc pas nécessaire d'utiliser une boucle, à l'instar de NumPy, même pour des logiques plus complexes :

```
def myfunction(l):
    if l["Genre"] == "homme":
        return l["Age"]+5
    else:
        return l["Age"]+2
df["Age new"] = df.apply(myfunction, axis=1)
```

Extraction de lignes. La sélection de lignes répondant à certains critères est très importante et assez simple, par exemple pour les personnes âgées de plus de 30 ans :

```
df[df["Age"]>30]
```

	Nom	Age	Genre	Age inv	Age new
1	M. William Allen	35	homme	2.857143	40
2	Mme Elizabeth Bonnell	58	femme	1.724138	60

ou les femmes âgées de plus de 30 ans :

```
df[(df["Age"]>30) & (df["Genre"]=="femme")]
```

	Nom	Age	Genre	Age inv	Age new
2	Mme Elizabeth Bonnell	58	femme	1.724138	60

Ce concept est très similaire à l'indexation NumPy. Pour une sélection plus complexe, nous pourrions utiliser la syntaxe suivante :

```
value=3
df.query('`Age inv`>@value and Genre=="homme"')
```

	Nom	Age	Genre	Age inv	Age new
0	M. Harris Braund	22	homme	4.545455	27

Slicing. Comme pour NumPy, il est possible d'utiliser le slicing pour sélectionner une partie du tableau :

```
df.loc[0:1, "Genre":"Age new"]
```

	Genre	Age inv	Age new
0	homme	4.545455	27
1	homme	2.857143	40

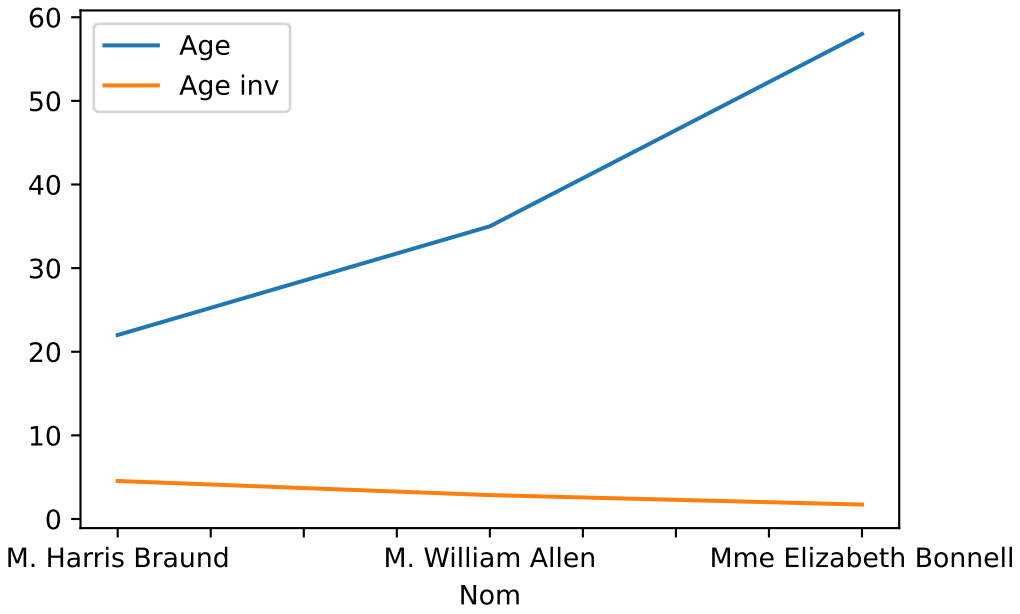
À remarquer que les points d'extrémité sont inclus, contrairement à la méthode standard de Python. Cependant, voici le slicing sans les points d'extrémité :

```
df.iloc[0:2,3:5]
```

	Age inv	Age new
0	4.545455	27
1	2.857143	40

Représentations graphiques. Enfin, les données peuvent être représentées graphiquement de différentes manières. La manière la plus simple est de faire :

```
df.plot(x="Nom", y=["Age","Age inv"])
```



a. Télécharger au format CSV les données de la Banque mondiale sur l'éducation disponibles à l'adresse : https://data.worldbank.org/topic/education.

b. En inspectant les fichiers précédents, extraire les données nécessaires afin d'obtenir une table à deux colonnes, la première contenant les codes des pays, la seconde les noms des pays.

> **Indication**
>
> Utiliser la fonction `read_csv` pour lire un fichier CSV avec Pandas.

c. En regardant la documentation des fonctions `rename` et `set_index`, renommer les étiquettes de colonnes en `code` et `country`, puis définir le code du pays comme index de ligne. Le but est que le nom du pays puisse ensuite être déterminé à partir de son code avec la fonction `loc["FRA"]`.

d. Déterminer le code du Zimbabwe avec le tableau précédent.

e. Déterminer, dans les données téléchargées précédemment, le code associé à la proportion de chômage ainsi que celui correspondant à la proportion de personnes ayant au moins un master.

> **Indication**
>
> Rechercher dans le fichier `Metadata_Indicator...` les chaînes « Unemployment » et « Master ».

f. Tracer l'évolution du taux de chômage en France.

> **Indication**
>
> Le fichier `API...` contient un en-tête et commence seulement à la ligne 5 ; utiliser l'option `skiprows=4` pour ignorer l'en-tête.

g. Écrire une fonction `time_plot(code, countries)` permettant de tracer l'évolution temporelle de l'indicateur `code` pour les pays de la liste `countries`. La tester sur le taux de chômage et la proportion de personnes ayant au moins un master pour différents pays.

Exercice 12.2 - Loi de Benford

La loi de Benford prédit que statistiquement dans une liste de nombres donnés, la probabilité qu'un de ces nombres commence par le chiffre 1 est plus importante que celle qu'il commence par le chiffre 9. Plus précisément la loi de Benford prédit que la probabilité qu'un nombre commence par le chiffre d est :

$$p_d = \log_{10}\left(1 + \frac{1}{d}\right),$$

où $\log_{10}$ désigne le logarithme en base 10. Il est possible de vérifier que la loi de Benford est la seule qui reste invariante par changement d'unités, c'est-à-dire qu'en multipliant les nombres de la liste par une constante, les probabilités précédentes restent inchangées.

a. Écrire une fonction `firstdigit(n)` qui pour un nombre `n` donné retourne son premier chiffre et une fonction `occurrences(liste)` qui retourne le nombre d'occurrences des premiers chiffres de `liste`.

> **Indication**
>
> Faire en sorte que la fonction `occurrences` fonctionne même si la liste contient des zéros en les ignorant.

b. Vérifier si la loi de Benford semble satisfaite pour la suite des nombres $(2^n)_{n\in\mathbb{N}}$ en comparant l'histogramme empirique avec la loi de Benford.

c. Vérifier si la loi de Benford semble satisfaite pour la suite des nombres $(3n+1)_{n\in\mathbb{N}}$.

d. En allant sur le site de l'INSEE à l'adresse : https://www.insee.fr/fr/statistiques/7631680, télécharger le fichier au format CSV contenant les données de la population française par sexe et âge regroupé (POP1A). Importer ces données pour avoir la population par code postal, sexe et tranche d'âge.

> **Indication**
>
> La documentation sur comment lire un fichier est disponible ici. Il est également possible d'utiliser Pandas.

e. Déterminer si la liste de toutes les populations par commune, sexe et âge suit la loi de Benford.

f. Sommer les données précédentes pour obtenir la liste des populations par commune et déterminer si elle suit la loi de Benford.

g. Refaire les deux questions précédentes mais avec le fichier de population POP1B avec les âges non regroupés et en utilisant Pandas.

h. !! En allant sur le site de l'INSEE ou autre, télécharger son jeu de données préféré et tester s'il suit la loi de Benford.

> **Indication**
>
> Utiliser par exemple les comptes détaillés de l'État disponibles ici.

Exercice 12.3 - Méthode des moindres carrés

a. Reprendre les données de l'INSEE de la population française par sexe et âge POP1B. Écrire une fonction `pop(code)` qui retourne sous forme de liste ou de vecteur NumPy la population par âge, sans distinction homme-femme, dans la commune de code postal `code`. Utiliser cette

fonction pour déterminer le nombre total de personnes habitant dans la commune de code postal 75102.

b. Écrire une fonction `plot_pop(code)` affichant graphiquement les fractions de la population (normalisées par la population totale de la commune) en fonction de l'âge, sans distinction homme-femme, dans la commune de code postal `code`. Tester pour les communes de code postal 13201 et 75102.

Pour une commune donnée, si on note $p(a)$ la fraction de la population ayant l'âge a, on cherche les coefficients r_0, r_1, r_2 tels que la loi :

$$p(a) = r_0 + r_1 a + r_2 a^2 \,,$$

soit la mieux satisfaite pour les âges $a \geq 25$. Pour cela, on résout le problème aux moindres carrés :

$$\min_{r \in \mathbb{R}^3} \|Xr - p\|^2 \,,$$

où $a = (25, 26, \dots, 100)$ est le vecteur des âges, X est la matrice 76×3 telle que $X_{i,1} = 1, X_{i,2} = a_i$, $X_{i,3} = a_i^2$ et p est le vecteur des populations par âge $p_i = p(a_i)$. La solution de ce problème est $r = (r_0, r_1, r_2) \in \mathbb{R}^3$. Cette solution satisfait l'équation :

$$X^\mathsf{T} X r = X^\mathsf{T} p \,,$$

où X^T désigne la transposée de X.

c. Pour la commune 13201, former la matrice X et le vecteur p, puis déterminer la solution r.

d. Pour les communes de code postal 13201 et 75102, tracer la courbe théorique $r_0 + r_1 a + r_2 a^2$ en fonction de l'âge par-dessus les données.

Exercice 12.4 - Reconnaissance de chiffres manuscrits

Le but de cet exercice est de classifier des images de chiffres manuscrits, c'est-à-dire de reconnaître des chiffres écrit à la main. Il s'agit d'un exemple simple d'apprentissage automatique et une de ces premières applications industrielles à la lecture automatique des chèques ou des codes postaux.

Le jeu de données de chiffres manuscrits scannés provient du UCI ML repository. Pour importer ce jeu de données, il y a deux possibilités. Si le paquet scikit-learn (nom d'import `sklearn`) est installé, il suffit d'exécuter :

```
from sklearn.datasets import load_digits
digits = load_digits()
X, y = digits.data, digits.target
```

Dans le cas où le module `sklearn` est indisponible, il est possible utiliser les commandes suivantes à la place pour charger les données :

```
import urllib.request, gzip, io
# lien à télécharger
url =
 ↪  "https://raw.githubusercontent.com/scikit-learn/scikit-learn/main/sklearn/datasets/dat
# télécharge le fichier gz
file = urllib.request.urlopen(url)
# extrait le fichier gz
file = gzip.GzipFile(fileobj=io.BytesIO(file.read()))
# importe le fichier txt
digits = np.loadtxt(file, delimiter =',')
# extrait les images et les labels
X, y = digits[:,:-1], digits[:,-1]
```

Dans les deux cas, `X` est un tableau NumPy qui contient de nombreux exemples de chiffres manuscrits, numérisés en image de 8×8 pixels et stockés sous forme de tableaux de 64 nombres flottants. La variable `y` contient l'entier entre 0 et 9 correspondant au chiffre numérisé. On parle de *label*.

a. Déterminer les dimensions de `X` et `y` puis en déduire le nombre d'exemples contenus dans la base de données.

b. Afficher les données contenues dans `X` associées à l'indice `idx=12`. Il s'agit donc de la douzième ligne du tableau `X` en commençant la numérotation à zéro.

c. À l'aide des fonctions `reshape` de NumPy et `imshow` de Matplotlib, afficher l'image d'indice `idx=12`. Il est possible d'utiliser l'argument `cmap='gray'` dans l'appel de `imshow` pour afficher le résultat en niveau de gris. Quel chiffre est ainsi codé ?

Pour chacune des classes de chiffre (de 0 à 9), l'idée est de calculer son centroïde, *i.e.* la représentation « moyenne » d'une classe.

d. Définir les sous-tableaux de `X` et de `y` correspondant à tous les chiffres 0 numérisés.

e. Pour l'ensemble des 0 de la question précédente, calculer pour chaque pixel la valeur moyenne, afin de définir le « zéro moyen ».

f. Pour l'ensemble des chiffres de 0 à 9 tracer sur une même ligne l'image moyenne associée comme sur la Figure 12.1.

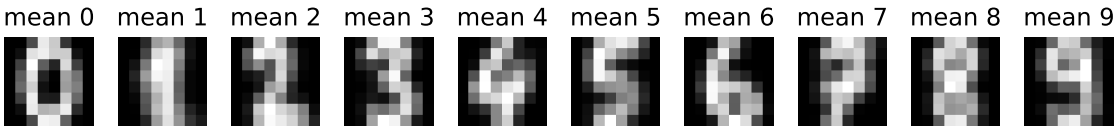


FIGURE 12.1 : Moyennes des chiffres numérisés.

Indication

Utiliser la fonction `subplot` de Matplotlib.

Finalement, nous allons implémenter notre propre classifieur : pour une nouvelle image de chiffre numérisé, on prédit la classe dont le chiffre moyen est le plus proche. Pour cela on partage notre jeu de données en deux parties de tailles semblables : la première partie servira de données d'entraînement (`X_train` et `y_train`) ; la seconde partie servira de données de tests (`X_test` et `y_test`).

g. Définir les variables : `X_train`, `y_train`, `X_test` et `y_test`.

h. Pour chaque chiffre de l'ensemble d'entraînement, calculer les centroïdes (c'est-à-dire les chiffres moyens) des classes de 0 à 9. On notera la variable contenant l'ensemble des moyennes `centroids_train`.

i. Écrire une fonction qui, donné un chiffre de l'ensemble de test (`X_test`), retourne le centroïde de `centroids_train` le plus proche dans la norme euclidienne.

j. Finalement, évaluer si le chiffre ainsi obtenu correspond au vrai chiffre en utilisant `y_test` et en déduire une estimation du pourcentage de bonnes prédictions sur l'ensemble de test.

Exercice 12.5 - Différentiation automatique !

Le but de cet exercice est d'aborder une des briques fondamentales de l'apprentissage automatique : la différentiation automatique. Il s'agit d'une technique permettant de calculer des dérivées ou gradients de fonctions Python de manière pratiquement transparente pour l'utilisateur. Donné une certaine fonction Python `f(x)`, le but de la différentiation automatique est qu'il soit presque aussi facile pour l'utilisateur d'évaluer la dérivée de `f` en un point `x=1` que de faire `f(1)`, et ce même si la fonction f est compliquée. Il s'agit de la brique fondamentale permettant à l'apprentissage automatique d'apprendre des paramètres en optimisant des fonctions non-linéaires compliquées.

L'idée de la différentiation automatique est d'utiliser la règle de dérivation des fonctions composées et la connaissance des dérivées des fonctions de bases. En effet, une fonction Python est « juste » une composition de fonctions (ou instructions) de bases.

Par simplicité, on s'intéresse ici uniquement à la composition de fonctions sur $\mathbb{R}$. Pour tout $i \in \{0, 1, \dots, n\}$, soit $f_i : \mathbb{R} \to \mathbb{R}$ une fonction élémentaire dont on connait la dérivée. On considère la composition des $i \leq n$ premières fonctions :

$$F_i = f_i \circ f_{i-1} \circ \cdots \circ f_1 \circ f_0 \,.$$

La dérivée est donnée par la règle de composition (ou règle de chaîne) :

$$F_i' = (f_i \circ F_{i-1})' = (f_i' \circ F_{i-1}) \cdot F_{i-1}' \,.$$

Cela donne une manière récursive de calculer F_n' avec l'ancrage $F_0' = f_0'$. Pour calculer la valeur de $F_n(x)$ pour un x donné, Pyhon va intrinsèquement calculer successivement $F_0(x)$, puis $F_1(x)$, $F_2(x)$, jusqu'à $F_n(x)$. La différentiation automatique consiste à évaluer en même temps ou ultérieurement $F_0'(x)$, $F_1'(x)$ jusqu'à $F_n'(x)$.

À noter que la différentiation automatique n'est pas une approximation numérique ni du calcul symbolique. En effet, la valeur de la dérivée est déterminée exactement (à la précision machine), ce qui n'est pas le cas avec une approximation numérique du type :

$$f'(x) \approx \frac{f(x+h) - f(x)}{h},$$

avec un certain $h > 0$ petit. Ce n'est pas non plus du calcul symbolique, car il n'y a aucun symbole dans la différentiation automatique, seulement des nombres réels. La différentiation automatique calcule la valeur de la dérivée en un point donné, alors que la différentiation symbolique le fait pour un symbole quelconque.

a. La première étape est de définir les dérivées des fonctions élémentaires. Pour cela, construire les tuples $\tilde{f} = (f, f')$ manuellement pour les fonctions élémentaires sin, cos, op : $x \mapsto -x$, inv : $x \mapsto x^{-1}$ et square : $x \mapsto x^2$.

b. Une composition de fonctions $F_n = f_n \circ f_{n-1} \circ \cdots \circ f_1 \circ f_0$, sera stockée en Python comme la liste de tuples $[\tilde{f_0}, \tilde{f_1}, \ldots, \tilde{f_n}]$. Définir ainsi en Python la composition correspondante à la fonction :

$$F(x) = \cos\left(\frac{1}{\sin(-\cos x^2)^2}\right).$$

c. Écrire une fonction `eval(liste, x)` qui donné une liste de tuples définissant une composition de fonctions F_n, retourne $F_n(x)$. Tester sur l'exemple précédent.

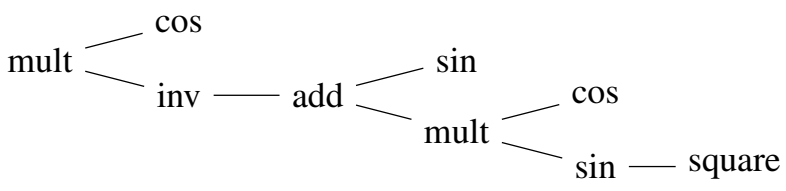
d. Écrire une fonction `autodiff(liste, x)` qui donné une liste de tuple définissant une composition de fonctions F_n, retourne $F_n(x)$ et $F_n'(x)$. Tester à nouveau sur le même exemple.

L'approche précédente ne permet que la composition de fonctions d'une variable, ce qui est très limitant, la somme et la multiplication étant des fonctions de deux variables. L'idée est de pouvoir également considérer des fonctions plus compliquées, comme par exemple :

$$G(x) = \frac{\cos(x)}{\sin(x) + \cos(x)\sin(x^2)}.$$

e. Implémenter sous forme de tuple comme précédemment les fonctions somme add : $x, y \mapsto x + y$ et multiplication mult : $x, y \mapsto xy$.

f. La fonction précédente G ne peut plus être représentée en Python sous la forme d'une liste de composition, car elle a la structure d'un graphe :



On choisit de la stocker en Python sous forme de liste de liste, avec comme premier élément de chaque liste la fonction à appliquer, les arguments étant les éléments suivants. Pour l'example G précédent :

```
myG = [mult, [cos], [inv, [add, [sin], [mult, [cos], [sin, [square]]]]]]
```

Écrire la composition simple précédente F sous cette nouvelle forme, ainsi que la fonction :

$$H(x) = \frac{\cos(x^2) + \sin x}{\cos x + 2\sin(x^{-1})}.$$

g. Écrire une fonction `eval(liste,x)` permettant d'évaluer au point `x` une expression sous forme de liste de listes, décrite précédemment. Tester sur les fonctions F, G et H.

h. !! Écrire une fonction `autodiff(liste,x)` qui en plus de retourner l'évaluation de la fonction en `x` retourne également l'évaluation de sa dérivée en `x`.

i. !! L'implémentation précédente n'est pas très utilisable concrètement. En pratique, la différentiation automatique se code en surchargeant les opérations de bases, dans le but d'être relativement transparente pour l'utilisateur. En utilisant le paquet JAX ou PyTorch, déterminer la dérivée de la fonction suivante par différentiation automatique en `x=0.4` :

```
def f(x):
    for i in range(50):
        if x>0.5:
            x = 3.7*x*(1-x)
        else:
            x = 3*x*(1-x)
    return x
```

Finalement, comparer avec la différentiation numérique.

Indication

Pour JAX, voir la documentation sur la différentiation automatique ici et, pour PyTorch, ici.

Exercice 12.6 - Réseau de neurones !

Le but de cet exercice est d'introduire la notation de réseau de neurones pour retrouver une fonction réelle dont on ne connaît que son évaluation bruitée. Plus précisément, on considère la fonction $f : [0,1] \to \mathbb{R}$ définie par :

$$f(x) = 1 + \sin(4\cos x)^2 ,$$

et on considère les données générées par 500 évaluations bruitées de f :

```
rng = np.random.default_rng(123456)
f = lambda x: 0.1 + np.sin(4*np.cos(x))**2
x = rng.random(500)
y = f(x) + rng.normal(0,0.1,500)
```

Le but est d'oublier que les données ont étés générées de cette façon et de retrouver une fonction approchant f uniquement à partir de x et y. Pour cela, un réseau de neurones à une couche sera utilisé. On parle d'apprentissage de la fonction à partir des données. Le principe est de construire la fonction apprise sous la forme :

$$f_\omega(x) = \frac{1}{n}\sum_{i=0}^{n-1} w_i \sigma(a_i x + b_i) ,$$

où $\omega = (a, b, w) \in \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n$ sont des paramètres à déterminer et σ la fonction sigmoïde :

$$\sigma(x) = \frac{1}{1+e^{-x}} .$$

Il s'agit d'un réseau de neurones à une couche avec n neurones. Le principe est de sommer n fonctions non-linéaires (les sigmoïdes) avec des poids d'entrée $a = (a_i)_{i=0}^{n-1}$, des biais $b = (b_i)_{i=0}^{n-1}$ et des poids de sortie $w = (w_i)_{i=0}^{n-1}$. Le choix des paramètres se fait de manière à minimiser la fonction de coût suivante :

$$J(\omega) = \sum_{k=0}^{499} \big(f_\omega(x_k) - y_k\big)^2 .$$

La stratégie pour minimiser J sur les paramètres ω est d'effectuer une descente de gradient en commençant avec des valeurs aléatoires des paramètres ω_0 puis en définissant successivement :

$$\omega_{i+1} = \omega_i - \eta J'(\omega_i) ,$$

où $J'(\omega)$ désigne le gradient de $J(\omega)$ par rapport aux paramètres, et $\eta \in (0,1]$ est un paramètre appelé taux d'apprentissage. L'idée de l'algorithme de descente de gradient est de faire bouger les

paramètres dans la direction de plus fort gradient afin de minimiser $J(\omega)$. On parle d'apprentissage des paramètres.

a. Représenter graphiquement les données `x` et `y` ainsi que la fonction f.

b. Déterminer la valeur de la fonction de coût J pour la fonction f :

$$J_f = \sum_{k=0}^{499} \big(f(x_k) - y_k\big)^2 .$$

c. Définir la fonction sigmoïde σ en Python ainsi que sa dérivée σ' et représenter la sigmoïde graphiquement.

d. Implémenter en Python une fonction `F(x,omega)` correspondante à la fonction $f_\omega(x)$. Faire en sorte que la fonction `F(x,omega)` soit vectorisée, c'est-à-dire si $x = (x_0, x_1, \dots, x_{k-1}) \in \mathbb{R}^k$ alors la fonction renvoie $(f_\omega(x_0), f_\omega(x_1), \dots, f_\omega(x_{k-1}))$.

e. Calculer à la main le gradient de $f_\omega(x)$ par rapport à ω (et non pas par rapport à x) et implémenter ce gradient en Python, en prenant soin qu'il soit également vectorisé.

f. Implémenter en Python $J(\omega)$ ainsi que son gradient $J'(\omega)$.

g. Avec le taux d'apprentissage $\eta = 0.01$ et quatre neurones, apprendre les paramètres $\omega \in \mathbb{R}^{12}$ qui tendent à minimiser J. Comparer la valeur de la fonction coût $J(\omega)$ de la fonction f_ω apprise avec la valeur de la fonction coût J_f de la fonction f. Représenter graphiquement la fonction f_ω en fonction de x pour ces paramètres et comparer avec la fonction f.

13 Cryptographie

Depuis le code de César, les méthodes cryptographiques permettant de transmettre des messages secrets ont évoluées en suivant les progrès permettant de les casser. La méthode Vigenère qui est une amélioration du code de César sera étudiée et nous verrons comment il est possible de casser cette méthode de chiffrement. Ensuite, la méthode de chiffrement RSA qui est une des méthodes de cryptographie asymétrique les plus utilisées actuellement sera introduite.

Concepts abordés

- code Vigenère
- plus grand diviseur commun
- importation de texte
- nombre premier et pseudo-premier
- petit théorème de Fermat
- algorithme d'Euclide
- algorithme de Miller-Rabin
- optimisation par décorateur
- chiffrement asymétrique RSA

Exercice 13.1 - Code de Vigenère

Le code de Vigenère consiste à choisir une clef formée par un mot secret (en majuscules) et à le transformer en un vecteur dont les éléments sont les positions de ces lettres dans l'alphabet. Par exemple, « ASECRET » correspond à (0, 18, 4, 2, 17, 4, 19). Pour coder un texte (en majuscules, sans espace ni ponctuation) avec la clef « ASECRET » il suffit de décaler la première lettre par 0, la deuxième par 18, la troisième par 4, et ainsi de suite et de recommencer en boucle. Les détails en particulier historiques sont disponibles sur Wikipédia.

a. Écrire une fonction `to_int(s)` qui transforme un caractère en sa place dans l'alphabet et écrire la fonction inverse `to_chr(i)`.

Indication

Voir la documentation des fonctions `ord` et `chr`.

b. Écrire une fonction `crypt(text, key)` qui chiffre `text` avec le mot secret `key`.

c. Écrire une fonction permettant de déchiffrer un texte en connaissant la clef.

Exercice 13.2 - Casser le code de Vigenère !

Charles Babbage a été le premier à casser le code de Vigenère. L'idée est que trois lettres consécutives apparaissant plusieurs fois dans le texte chiffré ont toutes les chances d'être la conséquence du chiffrement des mêmes lettres du message avec les mêmes lettres de la clef. Cela est encore plus probable avec un groupe de quatre lettres. Ainsi l'espacement entre deux mêmes groupes de lettres chiffrées est un multiple de la longueur de la clef. Par exemple, si la répétition d'un groupe est espacée de 28 lettres, puis celle d'un autre de 91, le plus grand commun diviseur de 28 et 91 est 7. Ainsi il est probable que la clef possède 7 lettres. Ensuite connaissant la taille de la clef, il suffit de se baser sur le fait que la lettre « E » est la plus courante en français. Pour que cette stratégie ait une chance de réussir, il faut que la taille du texte soit assez importante.

a. Écrire une fonction permettant de calculer le plus grand commun diviseur (PGCD) entre deux nombres. Écrire une autre fonction permettant de calculer le PGCD entre une liste de nombres.

b. Visiter le site du projet Gutenberg (https://www.gutenberg.org/browse/languages/fr), choisir son texte français préféré et le télécharger au format « Plain Text ». Écrire une fonction qui convertit le texte en majuscules et le débarrasse de toute la ponctuation et autres caractères spéciaux.

Indication

Pour convertir un texte en majuscules (en convertissant les accents) et supprimer tous les caractères de ponctuation et autres, il est possible d'utiliser la fonction suivante :

```
import unicodedata, re
def convert_upper(text):
    # convertir en majuscules
    text = text.upper()
    # convertir les accents
    text = unicodedata.normalize('NFKD', text)
    # supprimer les caractères spéciaux
    regex = re.compile('[^a-zA-Z]')
    text = regex.sub('', text)
    return text
```

c. Garder environ un millier de caractères au milieu du texte choisi et le chiffrer avec une clef. Écrire ensuite une fonction permettant de déterminer la longueur de la clef en regardant dans le message chiffré les chaînes identiques de trois caractères ou plus.

Indication

Former tout d'abord un dictionnaire avec comme clef toutes les occurrences de trois lettres et comme valeur les positions des occurrences. Ensuite déterminer la liste des distances entre les occurrences de trois lettres, puis calculer le PGCD de ces distances. Si ce PGCD est égal à 1 ou est trop petit, alors réessayer mais avec des chaînes de plus que trois caractères.

d. Écrire une fonction permettant de décrypter un message chiffré en retournant la clef. Essayer de décrypter le texte de son auteur favori avec cette fonction.

> **Indication**
>
> Pour trouver la première lettre de la clef, il faut calculer le nombre d'occurrences des 26 lettres de l'alphabet dans le message chiffré qui ont été chiffrées avec le premier caractère de la clef. En principe, la lettre dont l'occurrence est maximale correspond à la lettre « E ». Il suffit ensuite de faire la même chose pour trouver les autres lettres de la clef.

Exercice 13.3 - Générer des nombres premiers

La plupart des algorithmes de cryptage actuels sont fondés sur l'utilisation de grands nombres premiers. Le but est d'écrire une fonction permettant de générer des nombres premiers. La première étape est de générer un grand nombre aléatoire, c'est-à-dire ayant un certain nombre de bits. Ensuite un test de primalité permet de décider si ce nombre est premier ou pas. Si $\pi(n)$ désigne le nombre de nombres premiers inférieurs ou égaux à n, alors asymptotiquement $\pi(n) \approx \frac{n}{\ln n}$. Pour un nombre inférieur à n tiré au hasard la probabilité qu'il soit premier est d'environ $1/\ln(n)$. Par exemple pour générer un nombre premier de 1 024 bits (le minimum garantissant une sécurité raisonnable actuellement), c'est-à-dire de l'ordre de 2^{1024}, il faut essayer $\ln(2^{1024}) \approx 710$ nombres aléatoires avant d'en trouver un qui soit premier. Vu que tous les nombres pairs ne sont clairement pas premiers, il suffit d'en tester en moyenne 355.

a. Écrire un programme permettant de générer un nombre aléatoire impair de k bits, c'est-à-dire compris entre 2^{k-1} et 2^k.

> **Indication**
>
> La façon la plus rapide d'implémenter cela est d'utiliser les opérations sur les bits expliquées ici.

La façon la plus simple de déterminer si un nombre n est premier est d'essayer de le diviser par tous les nombres entiers $1 < d < n$. Il y a deux raisons qui permettent de ne pas tester tous les d entre 2 et $n-1$. La première est qu'il est inutile d'essayer les d pairs plus grands que 2. La seconde est qu'il est inutile de tester des nombres plus grands que $\sqrt{n}$.

b. Écrire un algorithme `isprime(n)` permettant de déterminer si un nombre est premier ou pas.

c. Écrire une fonction `generate(k,primality)` permettant de générer un nombre premier aléatoire de `k` bits avec le test de primalité `primality`. Tester avec le test de primalité `isprime`. Est-il raisonnable de pouvoir espérer générer un nombre premier de 1 024 bits avec cet algorithme ?

Exercice 13.4 - Générer des nombres pseudo-premiers

L'algorithme précédent permettant de générer des nombres premiers étant inutilisable pour générer de grands nombres premiers, une autre approche, probabiliste, est à préconiser. Un test de primalité probabiliste décide qu'un nombre est premier s'il est premier avec une probabilité très grande. Un tel nombre est dit pseudo-premier. Ainsi un test probabiliste peut se tromper et supposer qu'un nombre est premier alors qu'en fait il ne l'est pas.

Le test de primalité le plus simple est fondé sur le petit théorème de Fermat : si n est premier, alors $a^{n-1} = 1 \pmod{n}$ pour tout $1 \leq a \leq n-1$. Ainsi si on trouve un a tel que $a^{n-1} \neq 1 \pmod{n}$, alors n n'est pas premier. Le test de primalité de Fermat teste N valeurs de a choisies aléatoirement et si $a^{n-1} = 1 \pmod{n}$ pour ces N valeurs, alors il déclare que n est probablement premier. Les nombres de Carmichael ne sont pas premiers, mais satisfont $a^{n-1} = 1 \pmod{n}$ pour tout a premier avec n. Les premiers nombres de Carmichael sont 561, 1 105 et 1 729. Si n n'est pas un nombre de Carmichael, alors la probabilité que le test de Fermat se trompe est de 2^{-N}. En choisissant par exemple $N = 128$, on obtient une probabilité de se tromper inférieure à 3×10^{-39}.

a. Écrire une fonction implémentant le test de primalité de Fermat. Utiliser ce test pour générer des nombres premiers aléatoires.

> **Indication**
>
> Voir la documentation de la fonction `pow` pour une implémentation rapide. Si OpenSSL est installé sur votre ordinateur, il est facile de vérifier si un nombre est premier avec la commande `openssl prime 11` par exemple pour 11.

b. ! Améliorer la rapidité de l'algorithme précédent en testant d'abord si n est divisible par les nombres premiers inférieurs à 1 000 avant d'appliquer le test de Fermat.

Le test de primalité de Fermat permet de générer de grands nombres pseudo-premiers avec une bonne probabilité de tomber juste. Le problème principal vient du fait de l'existence des nombres de Carmichael qui sont exclus de cette probabilité. Le test de primalité de Miller-Rabin permet d'éviter ce problème.

c. !! Comprendre et implémenter la méthode Miller-Rabin expliquée en détails sur Wikipédia.

Exercice 13.5 - Chiffrement RSA

L'algorithme RSA, des initiales de Ronald Rivest, Adi Shamir et Leonard Adleman qui l'ont inventé en 1983, est un des algorithmes de cryptographie asymétrique les plus utilisés encore actuellement. Un chiffrement asymétrique permet de transmettre un message crypté à une personne A sans avoir eu auparavant besoin de transmettre une clef secrète à la personne B qui chiffre le message. La création par A d'une clef publique suffit à B pour chiffrer le message et pour que A puisse le déchiffrer avec sa clef privée. Il y a trois grandes étapes dans l'algorithme RSA :

Création des clefs. La personne A voulant recevoir un message secret choisit deux très grands nombres premiers p et q qu'elle garde secrets. Elle calcule ensuite $n = pq$ et l'indicatrice d'Euler $\varphi(n) = (p-1)(q-1)$ qui compte le nombre d'entiers compris entre 1 et n qui sont premiers avec n. Puis elle choisit un exposant de chiffrement e qui est premier avec $\varphi(n)$. La clef publique de la personne A est donnée par le couple (n, e). Finalement, la personne A calcule l'exposant de déchiffrement d qui est l'inverse de e modulo $\varphi(n)$, *i.e.* tel que $ed = 1 \pmod{\varphi(n)}$. La clef privée de A est (p, q, d).

Chiffrement du message. Pour chiffrer son message, la personne B le transforme tout d'abord en un nombre entier $M < n$. Le message chiffré est alors donné par :

$$C = M^e \pmod{n}.$$

Déchiffrement du message. Le message chiffré C est alors transmis à A. Pour le déchiffrer, A calcule :

$$M = C^d \pmod{n}$$

qui est à nouveau le message original.

Note

Les nombres premiers p et q doivent être vraiment aléatoires, sinon il est possible de deviner leurs valeurs. Les nombres aléatoires générés par le module `random` le sont avec l'algorithme de Mersenne Twister. Cet algorithme n'est pas considéré comme cryptographiquement sûr dans le sens où une observation d'environ un millier de nombres générés aléatoirement par cet algorithme suffit à prédire toutes les itérations futures. Pour générer des nombres aléatoires cryptographiquement sûrs il faudrait utiliser le module `secrets`.

a. Montrer mathématiquement que le message déchiffré correspond bien au message original.

Indication

Si $a = b \pmod{\varphi(n)}$ et M est premier avec n, alors $M^a = M^b \pmod{n}$.

b. À partir de e et $\varphi(n)$ donnés, écrire une fonction `bezout(e, phi)` permettant de déterminer d tel que $ed = 1 \pmod{\varphi(n)}$.

Indication

Utiliser l'algorithme d'Euclide généralisé qui permet de déterminer le PGCD g entre deux nombres a et b ainsi que x et y satisfaisant $ax + by = g$.

c. Écrire un algorithme `generate_keys(length)` qui génère des nombres premiers p et q tels que n ait `length` bits, puis détermine $\varphi(n)$, e et d et enfin retourne la clef publique (n, e) et la clef privée (p, q, d).

d.

En choisissant d'encoder chaque caractère sur 8 bits, une chaîne de caractères de longueur ℓ s'écrit comme une liste $(a_0, a_1, \dots, a_\ell)$ avec chaque $0 \leq a_i \leq 255$. Cette liste peut être convertie en un entier k de la façon suivante :

$$k = \sum_{i=0}^{\ell} a_i 256^i$$

Écrire une fonction `toint` et une fonction `tostr` permettant respectivement de convertir une chaîne de caractères en cet entier et inversement.

e. Écrire une fonction pour chiffrer un texte avec une clef publique et une autre permettant de le déchiffrer avec la clef privée. Pour cela il faut s'assurer que le texte soit convertible en un entier inférieur à n, sinon il faut le découper en blocs et les chiffrer séparément.

Exercice 13.6 - Casser le chiffrement RSA !!!

Voici une clef publique :

```
(681173384823993924634706123599189498672431584217436911278577378677214308161 93,
 ↪ 857480488977203937958496372891351154528533346142902255835809356706830819 3213)
```

et un message chiffré avec cette clef publique :

```
[384136230655609584943005932212261691823487492527778599508915777719208421582 70,
 ↪ 593153705940082564915922596418643741272947091362043824201026586345201986 00601,
 ↪ 582172413835618154216588121262493114040161906134436732040662985849533759 69428,
 ↪ 351666963259528733257189950383336582459201228650643717810909958607026037 43974,
 ↪ 346298136521733055129430697157070011595338654730304001200530708481165243 32508,
 ↪ 748664689684511299224831039211115241502455700261258855626936702383292767 2806,
 ↪ 125143443313597657595141266288931704883103648253653572477639726003265877 42385,
 ↪ 358433180418549489850663910465370138192579049518961018518047012874738815 8399,
 ↪ 478151924380871034234851485558604744021102404120358008352595867684967740 47047]
```

a. Décrypter le message précédent !

> **Indication**
>
> Il est probablement nécessaire de choisir un algorithme adapté, par exemple utilisant des cribles quadratiques (QS, MPQS, SIQS).