Programmation Python pour les mathématiques

Julien Guillod
Sorbonne Université

2019-2020

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. Ces exercices servent de base aux travaux pratiques 3MA100 donnés à Sorbonne Université en troisième année de licence.

Ce document est aussi disponible au format HTML et Jupyter Notebook. Les notebooks peuvent également être exécutés sur mybinder.

Les corrections et remarques concernant ce document sont les bienvenues à l'adresse *julien.guillod CHEZ sorbonne-universite.fr* en commençant le sujet du message par *[python]*.

Table des matières

1 Introduction **5**
- 1.1 Pourquoi Python ? . . . 5
- 1.2 Prérequis . . . 5
- 1.3 Documentation . . . 6
- 1.4 Installation simple . . . 6
- 1.5 Installation avancée . . . 7
- 1.6 Lancement de Jupyter Lab . . . 7
- 1.7 Utilisation de Jupyter Lab . . . 8

2 Structures de données **11**
- Exercice 2.1 : Listes . . . 11
- Exercice 2.2 : Tuples . . . 13
- Exercice 2.3 : Ensembles . . . 13
- Exercice 2.4 : Dictionnaires . . . 14

3 Structures homogènes **16**
- Exercice 3.1 : Introduction à Numpy . . . 16
- Exercice 3.2 : Opérations sur les tableaux . . . 18
- Exercice 3.3 : Représentations graphiques . . . 18
- Exercice 3.4 : ! Indexage de tableaux . . . 21

4 Intégration **23**
- Exercice 4.1 : Méthode des rectangles . . . 23
- Exercice 4.2 : Méthode des trapèzes . . . 24
- Exercice 4.3 : Méthode de Monte-Carlo . . . 25
- Exercice 4.4 : ! Méthode de Simpson . . . 25
- Exercice 4.5 : !! Module `scipy.integrate` . . . 26

5 Théorie des graphes **27**
- Exercice 5.1 : Graphes comme dictionnaires . . . 27
- Exercice 5.2 : Triangles dans un graphe . . . 28
- Exercice 5.3 : !! Module NetworkX . . . 29

6 Algèbre **30**
- Exercice 6.1 : Décomposition LU . . . 30
- Exercice 6.2 : Méthode de la puissance itérée . . . 31
- Exercice 6.3 : Groupes de permutations . . . 32

7 Zéro de fonctions **34**
Exercice 7.1 : Méthode de Newton en une dimension 34
Exercice 7.2 : Méthode de Newton en plusieurs dimensions 35
Exercice 7.3 : Attracteur de la méthode de Newton 35
Exercice 7.4 : !! Équation différentielle non-linéaire 36

8 Probabilités et Statistiques **38**
Exercice 8.1 : Loi de Benford 38
Exercice 8.2 : Ruine du joueur 39
Exercice 8.3 : Percolation 40

9 Équations différentielles **42**
Exercice 9.1 : Méthodes d'Euler 42
Exercice 9.2 : Méthodes de Runge-Kutta 44
Exercice 9.3 : Mouvement d'une planète 45
Exercice 9.4 : !! Équation des ondes cubique 46
Exercice 9.5 : !!! Méthodes de Bogacki-Shampine 47

10 Calcul symbolique **49**
Exercice 10.1 : Introduction à Sympy 50
Exercice 10.2 : Fonction pathologique 51
Exercice 10.3 : ! Fonction de Green du laplacien 51

11 Cryptographie **54**
Exercice 11.1 : Code de Vigenère 54
Exercice 11.2 : ! Casser le code de Vigenère 55
Exercice 11.3 : Générer des nombres premiers 56
Exercice 11.4 : Générer des nombres pseudo-premiers 56
Exercice 11.5 : Chiffrement RSA 57
Exercice 11.6 : !!! Casser le chiffrement RSA 58

1 Introduction

Le but des exercices de ce recueil n'est pas d'apprendre la syntaxe du langage Python ni ses subtilités, mais de se focaliser sur son utilisation 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. La résolution des exercices proposés devrait permettre d'avoir une bonne vision d'ensemble de l'utilisation de la programmation par les mathématiques et d'être à même de résoudre des problèmes mathématiques complexes avec l'aide de la programmation. Les exercices plus difficiles sont indiqués par des points d'exclamations :

- ! : plus long ou plus difficile ;
- !! : passablement plus long et complexe ;
- !!! : défi.

1.1 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, par contre il permet une grande rapidité de développement qui permet au programmeur de travailler un peu moins au détriment de l'ordinateur qui devra travailler un peu plus. Cette particularité a fait que Python est devenu l'un des principaux langages de programmation utilisé par les scientifiques.

1.2 Prérequis

Les prérequis sont de connaître les bases du langage Python, par exemple telles qu'enseignées dans le cours 1I001 dispensé à Sorbonne Université. Les personnes ne connaissant pas bien les bases du langage Python sont fortement incitées à suivre le MOOC (Massive Open Online Course) *Python 3 : des fondamentaux aux concepts avancés du langage* disponible sur le site de France Université Numérique. En dehors des périodes d'ouverture du MOOC, les vidéos du MOOC sont disponibles sur YouTube.

Finalement les exercices demandent d'avoir accès à un ordinateur disposant de Python 3.6 (ou plus récent) contenant les modules suivants : `numpy`, `scipy`, `sympy`, `matplotlib`. Un éditeur de code permettant l'écriture en Python est aussi vivement conseillé. Il est ici suggéré d'utiliser Jupyter Lab, car il permet à la fois l'écriture des notebooks interactifs et des scripts. 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.

1.3 Documentation

Il n'est généralement pas utile (et 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.

1.4 Installation simple

La façon la plus simple d'installer Python 3.6 (ou plus récent) et toutes les dépendances nécessaires est d'installer Anaconda. Les procédures d'installations détaillées selon chaque système d'exploitation sont décrites à l'adresse : `https://docs.anaconda.com/anaconda/install/`. Les procédures suivantes sont un résumé rapide de la procédure d'installation.

Installation sous Windows.

1. Télécharger Anaconda pour Python 3.6 (ou plus récent) à l'adresse : `https://www.anaconda.com/download/#windows` ;
2. Double cliquer sur le fichier téléchargé pour lancer l'installation d'Anaconda, puis suivre la procédure d'installation (il n'est pas nécessaire d'installer VS Code).
3. Une fois l'installation terminée, lancer Anaconda Navigator à partir du menu démarrer.

Installation sous macOS.

1. Télécharger Anaconda pour Python 3.6 (ou plus récent) à l'adresse : `https://www.anaconda.com/download/#macos` ;
2. Double cliquer sur le fichier téléchargé pour lancer l'installation d'Anaconda, puis suivre la procédure d'installation (il n'est pas nécessaire d'installer VS Code).
3. Une fois l'installation terminée, lancer Anaconda Navigator à partir de la liste des applications.

Installation sous Linux.

1. Télécharger Anaconda pour Python 3.6 (ou plus récent) à l'adresse : `https://www.anaconda.com/download/#linux` ;
2. Exécuter le fichier téléchargé avec `bash` puis suivre la procédure d'installation (il n'est pas nécessaire d'installer VS Code).
3. Une fois l'installation terminée, taper `anaconda-navigator` dans un nouveau terminal pour lancer Anaconda Navigator (ou taper `source ~/.bashrc`).

1.5 Installation avancée

La procédure suivante décrit l'installation manuelle de l'environnement Python qui sera utilisé. La procédure est décrite pour Ubuntu 18.04, mais est adaptable aux autres systèmes d'exploitation.

1. Installer Python et Pip :

Terminal
```
sudo apt install python3 pip3
```

2. Installer les modules `numpy`, `scipy`, `matplotlib`, `numba` et `sympy`. Il existe pour cela deux possibilités :

- Installer les modules provenant des paquets :

Terminal
```
sudo apt install python3-numpy
sudo apt install python3-scipy
sudo apt install python3-matplotlib
sudo apt install python3-numba
sudo apt install python3-sympy
```

- Utiliser Pip pour installer les modules :

Terminal
```
pip3 install numpy
pip3 install scipy
pip3 install matplotlib
pip3 install numba
pip3 install sympy
```

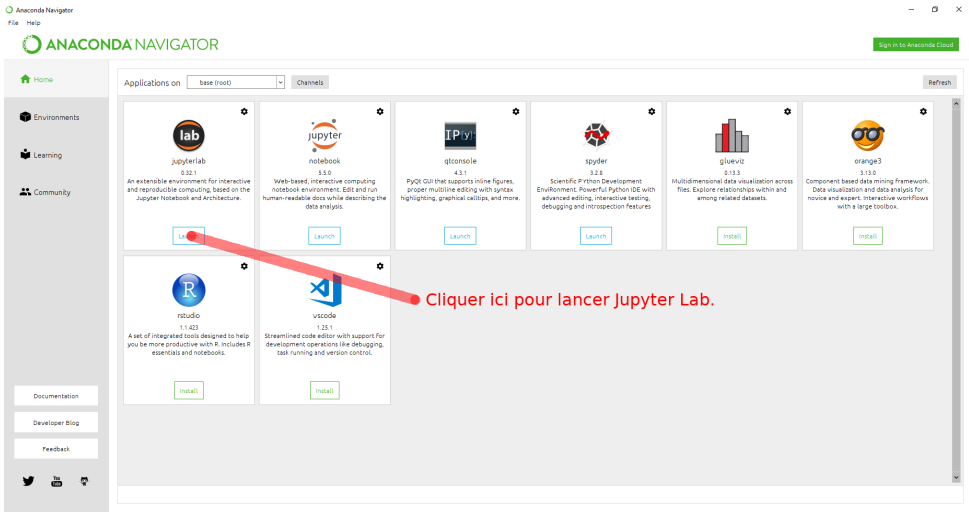
3. Installer Jupyter Lab avec Pip :

Terminal
```
pip3 install jupyterlab
```

Remarque. Suivant les systèmes d'exploitation il faut remplacer la commande `pip3` par `pip`. Si vous rencontrez un problème de permissions lors de l'exécution de ces commandes, il faut probablement rajouter `--user` à la fin de chacune d'entre elles.

1.6 Lancement de Jupyter Lab

Avec Anaconda Navigator. Si Anaconda a été installé, il suffit de lancer Anaconda Navigator et de cliquer sur l'icône "jupyterlab" :



En ligne de commande. Pour lancer Jupyter Lab en ligne de commande, il faut taper `jupyter lab` dans un terminal. Pour quitter, il faut fermer la fenêtre du navigateur, puis taper `Ctrl+C` suivi de `y` (en anglais) ou `o` (en français) dans le terminal où la commande `jupyter lab` a été exécutée.

Sur le bureau de l'UTES.

Avertissement
Disponible seulement pour les personnes disposant d'un accès informatique à Sorbonne Université.

Aller sur le site de l'UTES puis cliquer sur l'icône correspondante à votre système d'exploitation et suivez les instructions pour vous connecter au bureau à distance.

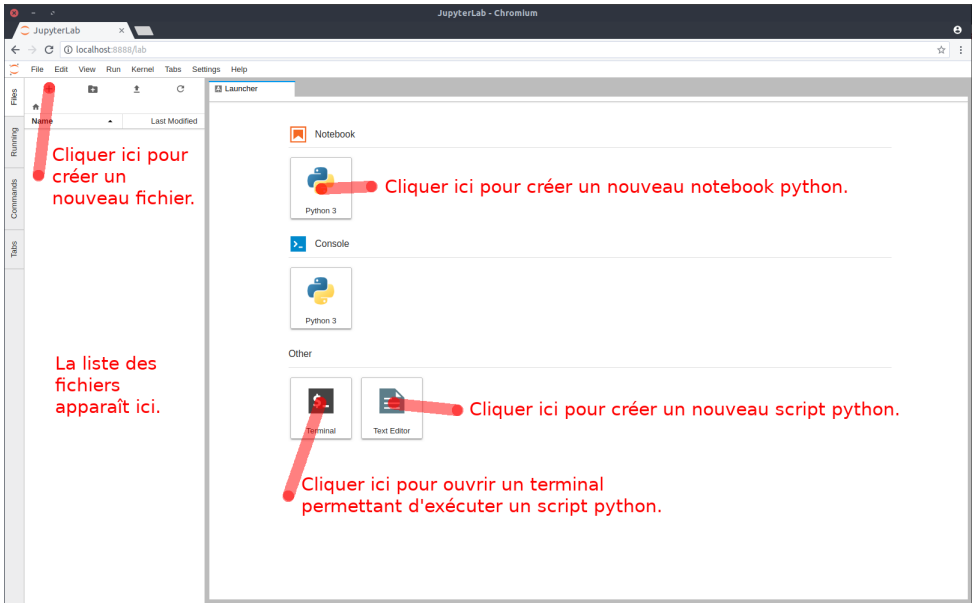
Pour lancer Jupyter Lab une fois connecté au bureau de l'UTES :

- cliquer sur la loupe située en haut ;
- taper "jupyter lab" dans le champs de rechercher suivi de la touche `ENTRÉE` ;
- cliquer sur "JUPYTER LAB POUR PYTHON 3.6" ;
- ne pas fermer la fenêtre noire avant d'avoir terminé de travailler sous Jupyter Lab.

En ligne avec mybinder. Pour les personnes ne pouvant ou ne voulant pas installer Python 3.6 et n'ayant pas accès au bureau de l'UTES, il est possible d'utiliser Jupyter Lab en ligne ici. Malheureusement, seulement cent personnes peuvent utiliser cette méthode en même temps. Les documents stockés sur mybinder.org sont automatiquement effacés donc il faut impérativement les sauvegarder sur votre propre ordinateur avant de quitter.

1.7 Utilisation de Jupyter Lab

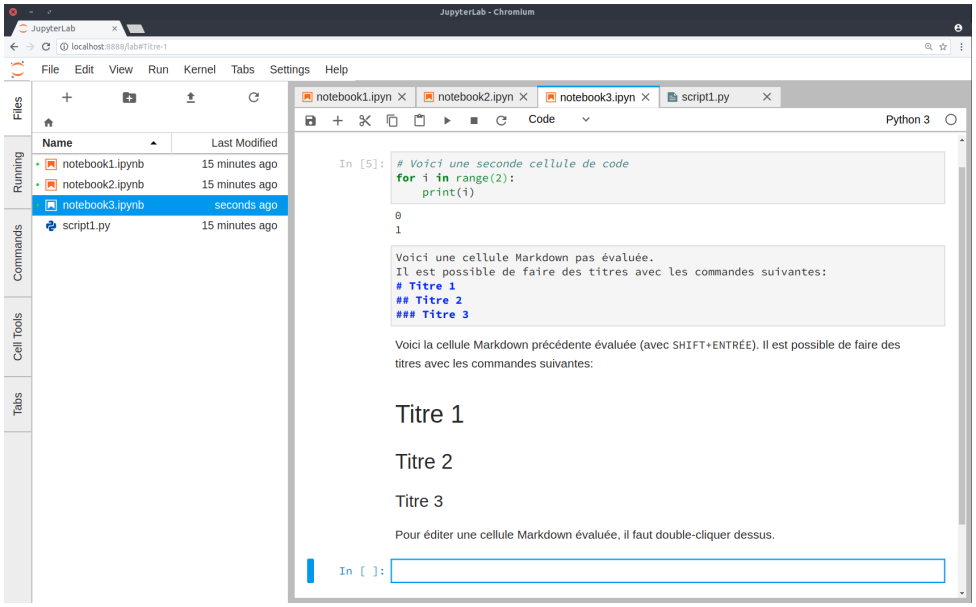
Une fois Jupyter Lab lancé, la fenêtre suivante doit apparaitre dans un navigateur :



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 peuvent être évaluées de manière interactive à la demande ce qui permet une grande flexibilité. Un script Python est simplement un fichier texte contenant des instructions Python. Un script Python s'exécute en entier de A à Z et il n'est pas possible d'interagir interactivement avec lui pendant son exécution (à moins que cela n'ai été explicitement programmé). Pour exécuter un script python il est nécessaire d'ouvrir un terminal.

Commandes de bases :

- Créer un nouveau fichier : cliquer sur le bouton `PLUS` 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 : combinaison des touches `SHIFT+ENTRÉE`.
- 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.

Remerciements : Merci à Marie Postel et Nicolas Lantos pour leurs relectures attentives de ce recueil et pour les nombreuses corrections et suggestions. Merci également à Cédric Boutillier et Cindy Guichard pour avoir pointés différentes erreurs.

2 Structures de données

Pour représenter des structures de données, Python propose quatre types de bases : les listes (type `list`), les tuples (type `tuple`), les ensembles (type `set`), et les dictionnaires (type `dict`). Le but est de voir les différences fondamentales entre ces différentes structures de données et de voir 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)
- type mutable et nonmutable
- 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, "chaine", True]
```

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

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

En particulier, il faut faire attention en copiant une liste. En exécutant le code suivant :

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

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
```

qui ne modifie pas `list0`. A 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)
```

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

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

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 par exemple si `a` est 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 :

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

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}.

Réponse : $u_{100} = v_{100} = 717897987691852588770249$

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 $K = 1000$ et diverses valeurs de $n_0 \in \{10, 100, 1000, 10000\}$, afficher les cinq dernières valeurs calculées, c'est-à-dire $(v_{K-4}, v_{K-3}, v_{K-2}, v_{K-1}, v_K)$.

Réponse : Les assertions suivantes sont vraies :

```
vk(10,1000) == [1, 4, 2, 1, 4]
vk(100,1000) == [2, 1, 4, 2, 1]
vk(1000,1000) == [1, 4, 2, 1, 4]
vk(10000,1000) == [4, 2, 1, 4, 2]
```

Exercice 2.2 : Tuples

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

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

Au contraire des listes, les tuples ne sont pas mutables. Il n'est pas possible de 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 que un tuple est hashable, c'est-à-dire qu'il peut être utilisé comme clé dans un dictionnaire.

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 tuple 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 est une structure permettant de stocker des éléments hétérogènes au sens mathématique de la théorie des ensembles :

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

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

```
if "chaine" in set0:
    print("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("chaine") # retire "chaine de l'ensemble
```

Par contre les ensembles 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: unhashable type: 'set'
```

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

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

Une chaine de caractère peut être transformée en ensemble :

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

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

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

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

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

b) Chercher dans la documentation comment faire l'intersection, l'union, et la différence de deux ensembles. Déterminer les nombres inférieurs ou égal à 100 qui sont pas 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]
```

Un dictionnaire peut être vu comme un tableau associatif associant à chaque clef une valeur. La liste des clefs et 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 orange 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
```

Bien que les dictionnaires soient mutables, les clefs qui le composent doivent être des objets hashables, c'est-à-dire immutables. Ainsi une liste ou un ensemble ne peut pas servir de clef dans un dictionnaire :

```
dict0[list0] = "test"
TypeError: unhashable type: 'list'
dict0[set0] = "retest"
TypeError: unhashable type: 'set'
```

Par contre 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)}
```

Une dernière 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)
```

a) Comment définir un dictionnaire vide ?

b) Comment concaténer plusieurs dictionnaire 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 `mots_position(mots, x, n)` qui retourne la liste des mots ayant le caractère `x` comme `n`-ième lettre.

Réponse : Par exemple `mots_position(mots,'e',4)` doit retourner la liste :

```
['Clémentine', 'Datte', 'Groseille', 'Pêche', 'Poire', 'Pomme', 'Prune']
```

d) En imaginant que la liste des mots soit très longue, alors à chaque évaluation de la fonction `mots_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 `mots_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 chaines de caractères). Cette particularité fait que les structures de données Python sont extrêmement flexibles, par contre 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
- représentations graphiques
- optimisation par parallélisation

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 4x2
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
```

et alors Numpy va déterminer lui-même le type et la taille du tableau. A 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
```

A noter que les indices commencent à zéro et non pas à un. 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
```

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))
```

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).

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 les transformations suivantes successivement :

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

Exercice 3.2 : Opérations sur les tableaux

Les opérations arithmétique de bases sur les tableaux Numpy sont effectuées éléments par éléments :

```
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
```

La plupart des fonctions mathématiques définies par Numpy (voir ici) sont aussi effectuées éléments par éléments :

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

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

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

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

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 1000 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)
```

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.

Réponse : Le temps d'exécution avec les tableaux Numpy devrait être approximativement 50 à 100 fois plus rapide qu'avec les listes !

Exercice 3.3 : Représentations graphiques

Le module `matplotlib` permet de faire des représentations graphiques très variées. Pour l'utiliser, il est d'usage de l'importer ainsi :

```
%matplotlib inline
import matplotlib.pyplot as plt
```

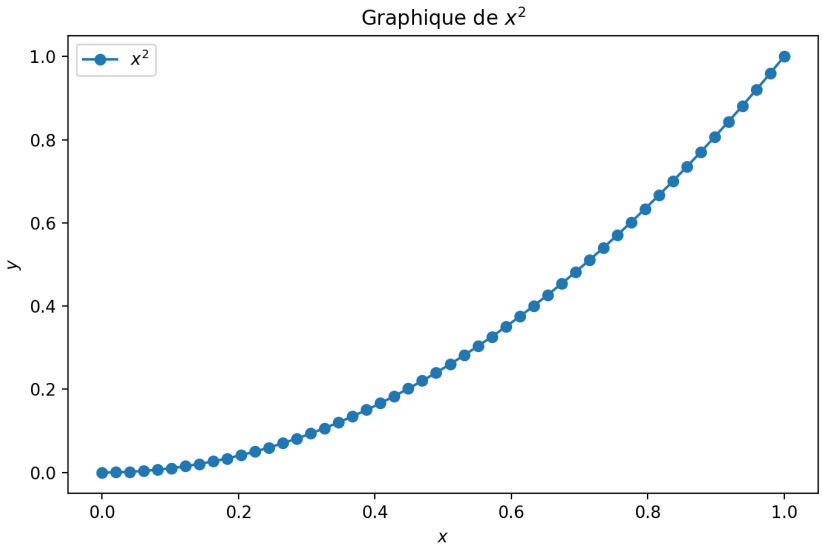
A noter que la première ligne permet de représenter les graphiques directement dans Jupyter Lab, mais n'est pas indispensable.

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()
```

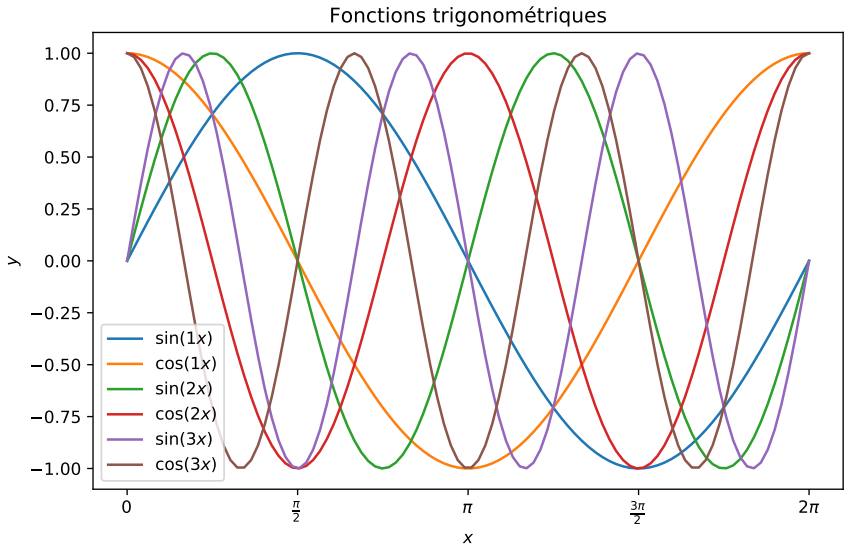
Afin de définir une jolie figure pouvant être exportée, la syntaxe est la suivante :

```
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()
```



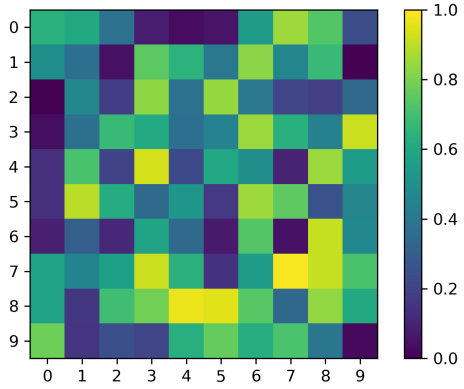
La documentation de Matplotlib est disponible ici.

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

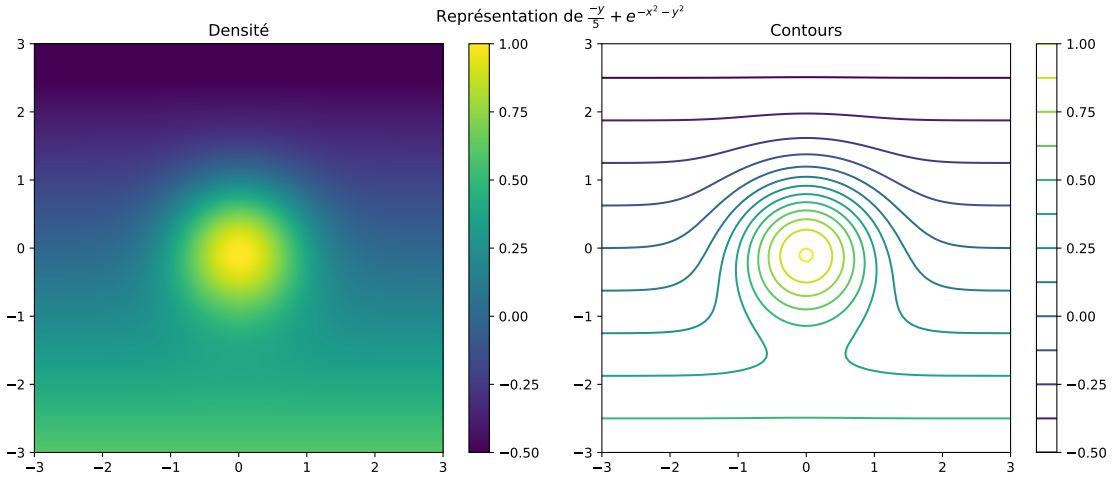


Indication : Utiliser la fonction `xticks` ou les fonctions `set_xticks` et `set_xticklabels` décrites ici.

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 :



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 :



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

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
```

A 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 celui qui l'indexe :

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

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
```

Finalement, il est possible d'indexer un tableau pour 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]
```

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

a) Les mesures étant censées être positives, modifier les données pour mettre zéro 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 suivante n'existe pas remplacer celle-ci par `np.nan`.

Intégration 4

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 de J.

Concepts abordés

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

Exercice 4.1 : 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.* $x_n = a + n\delta$. 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.

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 dans la documentation ici. Par exemple pour calculer e^π :

```
import math
math.exp(math.pi)
```

A noter que Numpy définit également des fonctions mathématiques, voir 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 4.2 : Méthode des trapèzes

La méthodes 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 4.3 : 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 :

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

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 de J par la méthode de Monte-Carlo.

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

b) Modifier la fonction précédente, pour qu'elle retourne en plus de la moyenne J_N également la variance empirique

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

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.

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 4.4 : ! 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 un en $x = a$ et la valeur zéro en $x = b$ et $x = c$. Faire une combinaison linéaire de trois polynômes de ce type.

b) Calculer $J_n \approx \int_{x_n}^{x_{n+1}} p_n(x)\mathrm{d}x$.

Indication : Il est possible de calculer cette intégrale à la main ou bien de le faire avec le module `sympy.integrate`, voir la documentation ici.

c) Simplifier à la main la somme des J_n approchés.

Réponse : Le résultat est

$$J \approx \frac{\delta}{3}\left[\frac{f(b)-f(a)}{2} + \sum_{n=0}^{N-1}\left(f(x_n) + 2f\left(\frac{x_n + x_{n+1}}{2}\right)\right)\right].$$

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 des méthodes des rectangles, des trapèzes et de Simpson en fonction de N.

Exercice 4.5 : !! Module `scipy.integrate`

Les méthodes d'intégrations précédentes et d'autres sont définies dans le module `scipy.integrate`. Ce module permet en particulier de traiter des cas plus compliqués, par exemple calculer

$$E_k(x) = \int_0^\infty \frac{e^{-xt}}{t^k}\mathrm{d}t\,.$$

à l'aide du code suivant :

```
import math, scipy.integrate
def integrand(t, n, x):
    return math.exp(-x*t)/t**n
def E(n, x):
    return scipy.integrate.quad(integrand, 1, math.inf, args=(n, x))[0]
E(4,2)
```

a) Lire la documentation de de module disponible ici.

Théorie des graphes 5

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, c'est-à-dire telle 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 $X = 1, 2, \ldots, n$ ayant comme arêtes les parties à deux éléments de X. En particulier, $K_4 = (X, E)$ où $X = \{1, 2, 3, 4\}$ et $E = \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}, \{3, 4\}\}$.

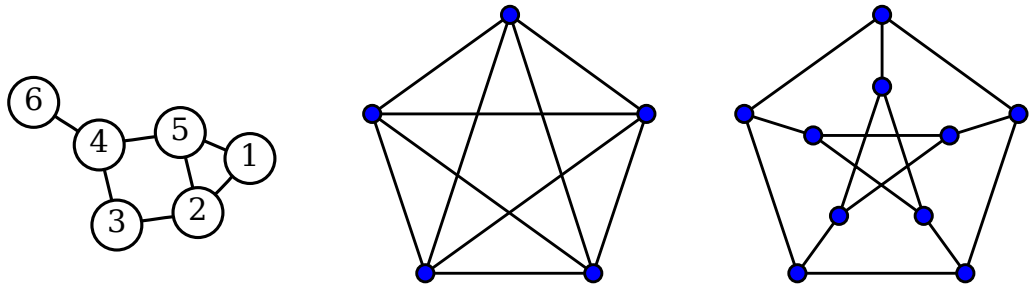
Concepts abordés

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

Exercice 5.1 : Graphes comme dictionnaires

Une façon de représenter un graphe G est de le faire avec un dictionnaire dont les clés sont les sommets, et la valeur associée à chaque clé $x \in X$ 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 clé 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 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 5.2 : 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 est importante pour comprendre sa structure.

a) Déterminer 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.

A 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 5.3 : !! 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 téléchargeable ici.

Algèbre 6

Tout d'abord une méthode de résolution de système linéaire par un algorithme direct est étudié, puis ensuite une méthode itérative sera utilisée pour déterminer le vecteur propre associé à la plus grande valeur propre d'une matrice. Finalement, 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 $A\boldsymbol{x} = \boldsymbol{b}$ en résolvant d'abord $L\boldsymbol{y} = \boldsymbol{b}$ puis $U\boldsymbol{x} = \boldsymbol{y}$. 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

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

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 :

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

puis la i-ème colonne de L par :

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

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

b) Écrire une fonction `LU_inplace(A)` qui ne crée par des 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.

c) Donné la décomposition LU d'une matrice A, écrire une fonction `solve(L,U,b)` qui résout le système linéaire $A\boldsymbol{x} = \boldsymbol{b}$. Faire également une version *in place* de l'algorithme.

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 sa plus grande valeur propre. Etant donné une matrice A de taille $n \times n$ et un vecteur $\boldsymbol{x}_0 \in \mathbb{R}^n$, la suite de vecteurs $(\boldsymbol{x}_k)_{k\in\mathbb{N}}$ est définie par :

$$\boldsymbol{x}_{k+1} = \frac{A\boldsymbol{x}_k}{\|A\boldsymbol{x}_k\|} .$$

En supposant que la matrice A soit diagonalisable avec des valeurs propres distinctes, alors il est relativement facile de montrer que la suite $(\boldsymbol{x}_k)_{k\in\mathbb{N}}$ converge vers le vecteur propre associé à la plus grande valeur propre de A.

Indication : La première étape est de remarquer que

$$\boldsymbol{x}_k = \frac{A^2\boldsymbol{x}_{k-2}}{\|A^2\boldsymbol{x}_{k-2}\|} = \cdots = \frac{A^k\boldsymbol{x}_0}{\|A^k\boldsymbol{x}_0\|} .$$

Vu que A est diagonalisable, soit $(\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_n)$ une base de vecteurs propres de A associés aux valeurs propres $\lambda_1, \lambda_2, \ldots, \lambda_n$. Sans perte de généralité, on suppose que λ_1 est la plus grande valeur propre. Le vecteur $\boldsymbol{x}_0$ se décompose dans la base :

$$\boldsymbol{x}_0 = \sum_{i=1}^{n} c_i \boldsymbol{v}_i ,$$

ainsi en supposant que $c_1 \neq 0$:

$$A^k \boldsymbol{x}_0 = \sum_{i=1}^{n} c_i \lambda_i^k \boldsymbol{v}_i = c_1 \lambda_1^k \left(\boldsymbol{v}_1 + \sum_{i=2}^{n} \frac{c_i}{c_1} \left(\frac{\lambda_i}{\lambda_1} \right)^k \boldsymbol{v}_i \right).$$

Vu que $\lambda_1 > \lambda_i$ pour $i \leq 2$, alors :

$$\lim_{k \to \infty} \left(\boldsymbol{v}_1 + \sum_{i=2}^{n} \frac{c_i}{c_1} \left(\frac{\lambda_i}{\lambda_1} \right)^k \boldsymbol{v}_i \right) = \boldsymbol{v}_1 \,,$$

et donc

$$\lim_{k \to \infty} \boldsymbol{x}_k = \lim_{k \to \infty} \frac{A^k \boldsymbol{x}_0}{\|A^k \boldsymbol{x}_0\|} = \frac{\boldsymbol{v}_1}{\|\boldsymbol{v}_1\|} \,.$$

Lorsque $c_1 \neq 0$, la suite $(\boldsymbol{x}_k)_{k \in \mathbb{N}}$ converge vers le vecteur propre normalisé associé à la plus grande valeur propre.

a) Définir une fonction `puissance(A, x0, k)` qui retourne $\boldsymbol{x}_k$. Avec cette fonction, déterminer le plus grand vecteur propre de la matrice

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

Réponse : Le plus grand vecteur propre est $\pm(0.70710678, 0.70710678)$

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

c) Écrire un programme permettant de calculer automatiquement la plus grande valeur propre et le vecteur propre associé d'une matrice carrée avec une certaine précision donnée.

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

e) Comparer la rapidité du code précédent et des fonctions Numpy pour des matrices de tailles $n \times n$ avec $n = 10, 100, 1000$.

Exercice 6.3 : Groupes de permutations

Le but est d'étudier les groupes de permutations en développant des algorithmes pour les caractériser. Un groupe de permutation G est généré par un certain nombre de permutations : $G = \langle g_1, g_2, \ldots, g_k \rangle$. 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`. A 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 $G = \langle g_1, g_2, \ldots, g_k \rangle$ engendré des permutations, peut être représenté par une liste de permutations `G = [g1,g2,...,gk]`. Écrire une fonction `orbit(G,x)` qui retourne l'orbite de $x \in G$:

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

Indication : Ne pas calculer l'ensemble des éléments de groupes, cela fait une liste beaucoup trop grande. Construire la liste $(X^0, X^1, \ldots, X^N)$ 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 de $x \in G$:

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

sous la forme d'un ensemble de générateurs.

Indication : Utiliser le théorème ou lemme de Schreier.

7 Zéro de fonctions

Le but de cette série d'exercices est de déterminer les zéros de manière approchéé 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 que 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 7.1 : 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 donné une fonction F, sa dérivée F' et une valeur initiale x_0 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 $e^{-x} = x$.

Réponse : La solution est approximativement donnée par $x = 0.56714$.

c) Sans utiliser la fonction `sqrt` 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 choisi.

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 7.2 : 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 $F(\boldsymbol{x}) = \mathbf{0}$ où $F : \mathbb{R}^n \to \mathbb{R}^n$ est une fonction assez régulière. Conceptuellement la méthode est identique : l'équation $F(\boldsymbol{x}) = \mathbf{0}$ est équivalente à $G(\boldsymbol{x}) = \boldsymbol{x}$ où :

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

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

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i - \big(F'(\boldsymbol{x}_i)\big)^{-1} F(\boldsymbol{x}_i)$$

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

Indication : Faire attention que les listes et les tableaux sont mutables. Pour avoir une performance optimale, il ne faut pas inverser la matrice jacobienne mais résoudre un système linéaire avec $F(\boldsymbol{x}_i)$ comme second membre.

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

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

Réponse : La solution est approximativement donnée par $x = 0.58853$ et $y = 0.98226$.

Exercice 7.3 : Attracteur de la méthode de Newton

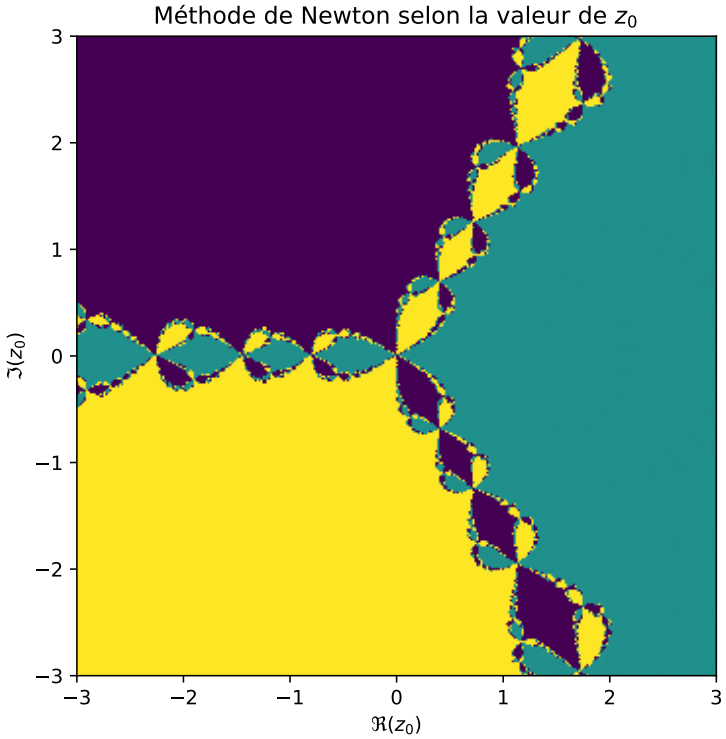
Le but de cet exercice est de résoudre l'équation $z^3 = 1$ dans le plan complexe à l'aide de la méthode de Newton et d'analyser vers laquelle des trois racines de l'unité la méthode va converger suivant le choix du point initial z_0.

a) Si nécessaire adapter la fonction `newton1d` pour que celle-ci marche également pour des 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.

Indication : La fonction `meshgrid` de Numpy peut être utile pour construire la matrice correspondante à l'ensemble des z_0.

Réponse : Le graphique donnant la convergence de z_0 vers une des trois racines est le suivant :



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 7.4 : !! Équation différentielle non-linéaire

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

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

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

La méthode employée est les différences finies qui consistent à chercher les valeurs de u aux points $x_n = \frac{2\pi n}{N}$ pour $n = 0, 1, \ldots, N$. 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 $n = 1, 2, \ldots, N-1$. Cette équation peut être vue comme une équation du type $F(\boldsymbol{u}) = \mathbf{0}$ pour $\boldsymbol{u} = (u_n)_{n=0}^N$ 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 $[0, 1]$ 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 $F(\boldsymbol{u}) = \mathbf{0}$.

Indication : Pour avoir une implémentation rapide, il est impératif de ne pas utiliser une boucle pour construire F mais d'utiliser le slicing Numpy.

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

Indication : La jacobienne est la dérivée de $F(\boldsymbol{u}) = F(u_0, u_1, \ldots, u_N)$ par rapport à $\boldsymbol{u} = (u_0, u_1, \ldots, u_N)$, c'est à dire :

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

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.

8 Probabilités et Statistiques

Premièrement la statistique de la proportion de nombres commençant par un certain chiffre sera étudiée. Finalement deux modèles probabilistes importants seront introduits.

Concepts abordés

- statistiques et probabilités
- importation de données
- diagrammes en bâtons
- optimisation par compilation
- marche aléatoire
- percolation
- transition de phase

Exercice 8.1 : 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'un nombre 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, *i.e.* en multipliant les nombres de la liste par une constante les probabilités précédentes restent inchangées.

a) Écrire une fonction `firstdigit(n)` qui donné un nombre `n` 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://insee.fr/fr/statistiques/3561090?sommaire=3561107`, télécharger le fichier au format TXT contenant les données de la population par sexe et âge (POP1A). Importer ces données pour avoir la population par code postal, sexe et âge.

Indication : La documentation sur comment lire un fichier est disponible ici. A noter que les fichiers TXT de l'INSEE sont encodés au format ISO-8859-1.

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

Exercice 8.2 : Ruine du joueur

Le but est de simuler l'évolution de la somme d'argent d'un joueur jouant à pile ou face. A chaque lancé le joueur gagne un franc 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 donné par une marche aléatoire :

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

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 $S_0 = k$. Représenter graphiquement plusieurs réalisations.

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

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 :

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

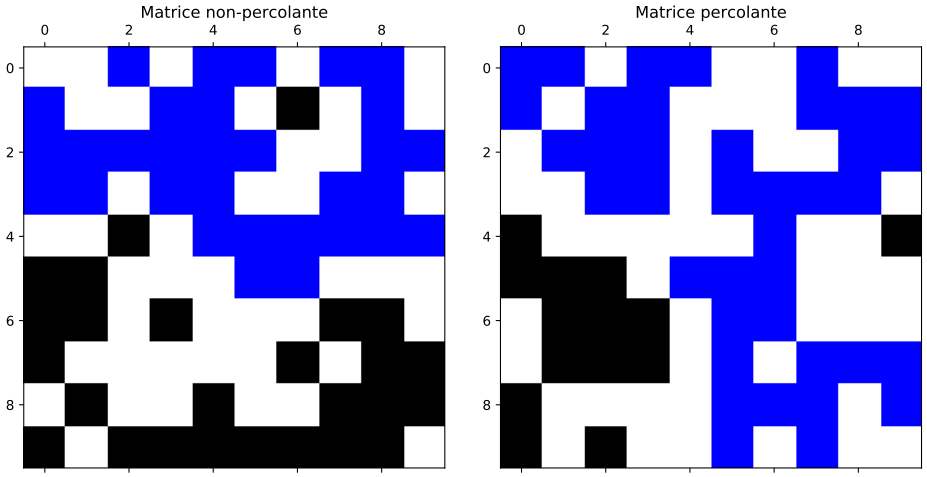
et le lieu de sortie :

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

où $\rho = q/p$.

Exercice 8.3 : 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 envahi par l'eau et ceux qui sont imperméables. Une matrice percole s'il existe un chemin d'eau allant de la ligne supérieur vers la ligne inférieur. Dans les exemples suivantes, les entrées d'une matrice pouvant être envahies par l'eau sont colorées et que les entrées effectivement remplies d'eau sont en bleu. La première matrice ne percole pas alors que la seconde oui :



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

Indication : La fonction `numpy.random.binomial` peut être utile.

b) Définir une fonction `fill(isopen)` qui donné une matrice de booléens 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) A 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-ci 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.

Réponse : Dans la limite des n très grands, une matrice percole presque surement si $p > 0.592746$ et presque jamais sinon.

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 votre 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`. A noter que Open MPI ou MPICH doit être installé sur votre ordinateur.

9 Équations différentielles

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

$$\dot{\boldsymbol{x}}(t) = f(t, \boldsymbol{x}(t))\,, \qquad\qquad \boldsymbol{x}(0) = \boldsymbol{x}_0\,,$$

où $f : \mathbb{R}^+ \times \mathbb{R}^n \to \mathbb{R}^n$ est une fonction assez régulière et $\boldsymbol{x}_0 \in \mathbb{R}^n$ une donnée initiale. A 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
- méthode adaptatives

Exercice 9.1 : 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 :

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

la seconde par différence finie arrière :

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

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

$$\frac{\boldsymbol{x}_{i+1} - \boldsymbol{x}_i}{t_{i+1} - t_i} = f(t_i, \boldsymbol{x}_i)$$

ce qui donne la formule d'Euler explicite :

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + (t_{i+1} - t_i) f(t_i, \boldsymbol{x}_i)\,.$$

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

$$\boldsymbol{x}_i = \boldsymbol{x}_{i-1} + (t_i - t_{i-1}) f(t_i, \boldsymbol{x}_i)\,.$$

La formule d'Euler explicite permet de calculer directement tous les $\boldsymbol{x}_i$ par récurrence en connaissant $\boldsymbol{x}_0$. Par contre la formule d'Euler implicite nécessite à chaque pas de temps, la résolution d'une équation non-linéaire pour $\boldsymbol{x}_i$ 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 $\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_m$ 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, la méthode d'Euler implicite peut en fait être rendue explicite en résolvant l'équation implicite à la main.

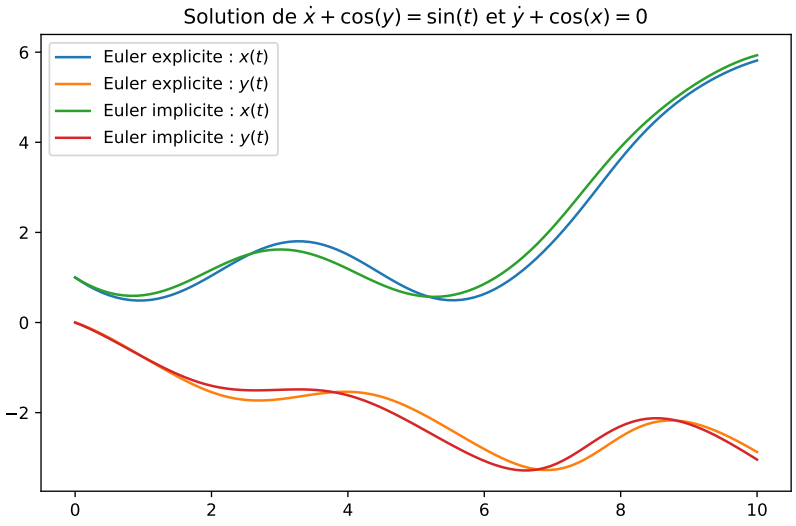
d) !! Définir une fonction `euler_implicit(f, Dxf, x0, t)` implémentant la méthode d'Euler implicite pour des équations non-linéaires. A noter que pour résoudre le problème non-linéaire avec la méthode de Newton, la dérivée de f selon $\boldsymbol{x}$ est nécessaire.

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

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

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

Réponse : Voici le graphique de la solution de l'équation différentielle avec un pas de temps $h = 0.1$:



Exercice 9.2 : 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 $\boldsymbol{x}_i$ donnant une approximation de la solution de $\dot{\boldsymbol{x}}(t) = f(t, \boldsymbol{x})$ au temps t_i pour $i \in \mathbb{N}$. Cette suite est définie par :

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + M(t_i, \boldsymbol{x}_i, t_{i+1} - t_i)\,,$$

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

$$M(t, \boldsymbol{x}, h) = hf(t, \boldsymbol{x})$$

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

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

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

$$M(t, \boldsymbol{x}, h) = \frac{h}{6}(\boldsymbol{k}_1 + 2\boldsymbol{k}_2 + 2\boldsymbol{k}_3 + \boldsymbol{k}_4)\,,$$

où

$$\begin{aligned}
\boldsymbol{k}_1 &= f(t, \boldsymbol{x})\,,\\
\boldsymbol{k}_2 &= f\left(t + \frac{h}{2}, \boldsymbol{x} + \frac{h}{2}\boldsymbol{k}_1\right),\\
\boldsymbol{k}_3 &= f\left(t + \frac{h}{2}, \boldsymbol{x} + \frac{h}{2}\boldsymbol{k}_2\right),\\
\boldsymbol{k}_4 &= f(t + h, \boldsymbol{x} + h\boldsymbol{k}_3)\,.
\end{aligned}$$

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

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

où

$$\begin{aligned}
\boldsymbol{k}_1 &= f(t, \boldsymbol{x}) \,, \\
\boldsymbol{k}_2 &= f(t + c_2 h, \boldsymbol{x} + h a_{21} \boldsymbol{k}_1) \,, \\
\boldsymbol{k}_3 &= f(t + c_3 h, \boldsymbol{x} + h(a_{31} \boldsymbol{k}_1 + a_{32} \boldsymbol{k}_2)) \,, \\
&\vdots \\
\boldsymbol{k}_s &= f(t + c_s h, \boldsymbol{x} + h(a_{s1} \boldsymbol{k}_1 + a_{s2} \boldsymbol{k}_2 + \cdots + a_{s,s-1} \boldsymbol{k}_{s-1})) \,.
\end{aligned}$$

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

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

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 donné une liste de temps $(t_i)_{i=0}^N$ retourne les valeurs correspondantes $\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_N$ 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 9.3 : 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 $\boldsymbol{x} \in \mathbb{R}^2$. L'étoile est supposée interagir avec la planète avec le potentiel :

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

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

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

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

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

c) A 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 $\boldsymbol{x}(t)$ dans le plan. Interpréter les résultats et en particulier pourquoi les cas $\alpha = -1$ et $\alpha = 2$ sont différents des autres.

Exercice 9.4 : !! Équation des ondes cubique

Le but est de résoudre numériquement l'équation des ondes nonlinéaire 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 $u : \mathbb{R}^+ \times \mathbb{R}$ avec $u_0, v_0 : \mathbb{R} \to \mathbb{R}$ deux fonctions données.

Remarque. A noter, que les propriétés de cette équation d'apparence très simple sont très mal comprises mathématiquement, voir l'article 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 7.4, montrer que l'équation peut s'approximer de la manière suivante :

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

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 :

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

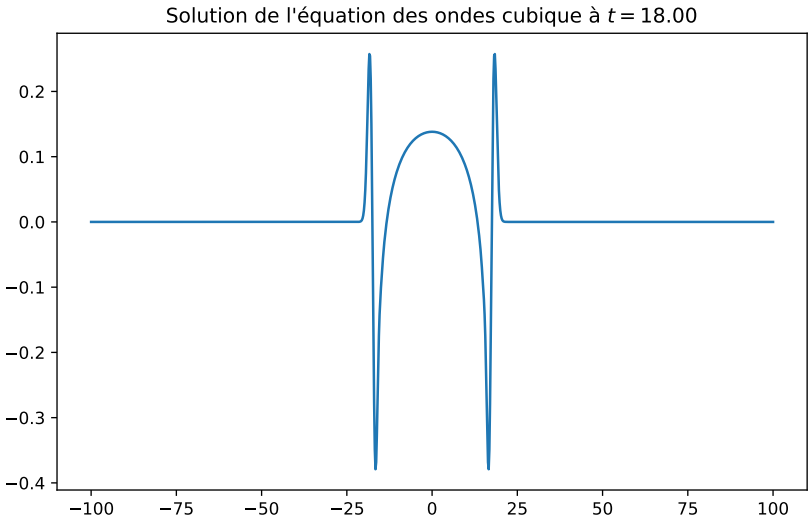
c) Déterminer la fonction $f : \mathbb{R} \times \mathbb{R}^{2N+2} \to \mathbb{R}^{2N+2}$ permettant de mettre l'approximation précédente sous la forme $\dot{\boldsymbol{u}} = f(t, \boldsymbol{u})$ pour $\boldsymbol{u} = (u, v)$ et implémenter cette fonction.

d) Résoudre l'équation différentielle donnée par $\dot{\boldsymbol{u}} = f(t, \boldsymbol{u})$ par exemple avec la méthode de Runge-Kutta d'ordre quatre. Un bon choix de paramètre est $L = 100$, $N = 1000$, pour la donnée initiale $u_0(x) = e^{-x^2}$ et $v_0(x) = 0$. A noter que la vitesse de propagation de l'onde est un et que donc après un temps plus grand que L l'onde sort de la boite $[-L, L]$ et donc ne correspond plus à une bonne approximation de l'équation initiale.

e) A l'aide du module `matplotlib.animation` réaliser une vidéo montrant l'évolution de l'onde en fonction du temps.

Indication : Utiliser par exemple la fonction `FFMpegWriter`.

Réponse : Voici la forme de la solution :



Exercice 9.5 : !!! Méthodes de Bogacki-Shampine

En combinant deux méthodes de Runge-Kutta d'ordres différentes (par exemple (2,3) ou (4,5)), il est possible d'obtenir 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 élevée (généralement $s - s$) est donnée par :

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

où les $\boldsymbol{k}_i$ sont identiques à ceux de la méthode d'ordre s. Une estimation de l'erreur est alors donnée par :

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

Une telle méthode est donnée par un tableau de Butchler 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 implémentées dans un module `nodepy` donc la documentation est disponible ici. Le nom de la méthode dans ce package est "BS5".

10 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 implémenté 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 Python comme un flottant :

```
import math
math.sqrt(8)
```

L'avantage de Sympy est que $\sqrt{8}$ est gardé en tant racine et même automatiquement simplifié :

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

A noter que la deuxième instruction, n'est pas nécessaire, mais permet de visualiser les résultats de manière plus jolie dans Jupyter Lab. Il est également assez courant d'importer toutes les fonctions de Sympy afin de ne pas avoir à mettre `sp.` devant chaque fonction :

```
from sympy import *
init_printing()
sqrt(8)
```

mais il faut alors faire attention aux éventuels conflits entre fonctions. La documentation de Sympy est disponible ici.

Concepts abordés

- symboles
- fonctions
- simplification
- analyse infinitésimale (limite, dérivation, intégration, développement limité)
- fonction pathologique
- fonction de Green
- coordonnées sphériques

Exercice 10.1 : Introduction à Sympy

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

```
x = sp.Symbol("x") # definit 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 et de simplifier des expressions :

```
expr = x + 2*y + e/4 + x**2 + 3*x + 2*y
sp.simplify(expr) # simplifie l'expression précédente
```

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)
```

Finalement il est possible de faire des substitutions :

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

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

Indication : La documentation sur les dérivées est disponible ici.

d) Lire la documentation de la fonction `integrate` et calculer les intégrales suivantes :

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

e) Calculer avec Sympy les limites suivantes :

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

f) 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.

g) 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 10.2 : Fonction pathologique

Le but est de construire une fonction qui a visuellement l'air d'être 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) A l'aide de Sympy calculer la fonction $g : \mathbb{R} \to \mathbb{R}$ définie en gardant les 100 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 la fonction 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 10.3 : ! Fonction de Green du laplacien

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

$$\Delta G(\boldsymbol{x}) = \delta(\boldsymbol{x}) \,,$$

où $\delta(\boldsymbol{x})$ est la distribution de Dirac.

Pour cela, nous introduisons les coordonnées sphériques $\boldsymbol{x}' = (r, \theta, \varphi)$ 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 :

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

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

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

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 $\mathbb{R}^3 \setminus \{\mathbf{0}\}$.

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 :

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

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 :

$$\lim_{r \to 0} \int_0^{\pi} \int_0^{2\pi} \left(\boldsymbol{\nabla} G(r) \cdot \boldsymbol{e}_r \right) J(r, \theta, \varphi) \, \mathrm{d}\varphi \mathrm{d}\theta = 1 \, .$$

h) Résoudre les équations sur les constantes d'intégrations et substituer dans la solution radial 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 :

$$\Delta f(\boldsymbol{x}) = g(\boldsymbol{x})$$

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

Indication : La solution est donnée par la formule de Green :

$$f(\boldsymbol{x}) = \int_{\mathbb{R}^3} G(\boldsymbol{x} - \boldsymbol{x}_0) g(\boldsymbol{x}_0) \, \mathrm{d}^3\boldsymbol{x}_0 = \int_{B_R} G(\boldsymbol{x} - \boldsymbol{x}_0) g(\boldsymbol{x}_0) \, \mathrm{d}^3\boldsymbol{x}_0 \,,$$

où R est choisi de sorte à ce que le support de g soit contenu dans B_R. Pour calculer le développement asymptotique de cette intégrale, la première étape est de convertir $G(\boldsymbol{x} - \boldsymbol{x}_0)$ en coordonnées sphériques pour $\boldsymbol{x}$ et $\boldsymbol{x}_0$. Vu que g est invariante par rotations selon l'axe vertical, alors en coordonnées sphérique g est indépendante de φ et est donné par $g(r, \theta)$. La seconde 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.

Cryptographie 11

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 divisieur commun
- importation de texte
- nombre premier et pseudo-premier
- petit théorème de Fermat
- algorithme d'Eucilde
- algorithme de Rabin-Miller
- optimisation par décorateur
- chiffrement asymétrique RSA

Exercice 11.1 : 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 seconde 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 11.2 : ! Casser le code de Vigenère

Charles Babbage a été le premier a 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 clé. 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 clé. Par exemple, si la répétition d'un groupe est espacée de 28 lettres, puis celle d'un autre de 91, le plus grand diviseur commun de 28 et 91 est 7. Ainsi il est probable que la clé 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 diviseur commun (PGDC) entre deux nombre. Écrire une autre fonction permettant de calculer le PGDC entre une liste de nombres.

b) Visiter le site du projet Gutenberg, choisir votre texte français préféré et télécharger-le au format "Plain Text". Écrire une fonction qui convertisse le texte en majuscules et le débarrasse de toute la ponctuation et autre.

Indication : Pour convertir un texte en majuscules (en convertissant les accents) et supprimer tous les caractères de ponctuations et autres, il est possible d'utiliser la fonction suivante :

```python
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 1000 caractères au milieu du texte choisi et chiffrer le 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 de 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 il réessayer mais avec des mots de longueur plus grande.

d) Écrire une fonction permettant de décrypter un message chiffré en retournant la clef. Essayer de décrypter le texte de votre 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 chiffé avec le premier caractère de la clef. La lettre ayant l'occurrence maximale doit correspondre à la lettre "E". Il suffit ensuite de faire la même chose pour trouver les autres lettres de la clef.

Exercice 11.3 : Générer des nombres premiers

La plupart des algorithmes de cryptage actuels sont basés sur l'utilisation de grands nombres premiers. Le but est d'écrire une fonction permettant de générer relativement rapidement des nombres dit pseudo-premiers, c'est-à-dire qui sont premiers avec une probabilité extrêmement grande. 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 $\leq n$ alors, asymptotiquement $\pi(n) \approx \frac{n}{\ln n}$. Ainsi en tirant au hasard un nombre inférieur à n la probabilité qu'il soit premier est d'environ $1/\ln(n)$. Par exemple pour générer un nombre premier de 1024 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 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 $1 < d < n$. Deux remarques 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 `primality=isprime`. Est-il raisonnable de pouvoir espérer générer un nombre premier de 1024 bit avec cet algorithme ?

Exercice 11.4 : Générer des nombres pseudo-premiers

L'algorithme précédent permettant de générer des nombres premiers étant inutilisable pour générer des 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 penser qu'un nombre est premier alors qu'en fait il ne l'est pas.

Le test de primalité le plus simple est basé 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 test 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 sont pas premiers, mais satisfont $a^{n-1} = 1 \pmod{n}$ pour tout a premier avec n. Les premiers nombres de Carmichael sont 561, 1105 et 1729. 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 du l'algorithme précédent en testant d'abord si n est divisible par les nombres premiers inférieurs à 1000 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 d'être effectivement premier. 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 se problème.

c) !! Comprendre et implémenter la méthode Rabin-Miller expliquée en détails sur Wikipédia.

Exercice 11.5 : Chiffrement RSA

L'algorithme RSA des noms de Ronald Rivest, Adi Shamir et Leonard Adleman qui l'ont inventé en 1983 est un des algorithmes de cryptographie asymétrique les plus utilisé 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 privé. 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'il garde secret. Ensuite elle calcule $n = pq$ 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 donné par le couple (n, e). Finalement, la personne A calcul 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.

Remarque

A noter que les nombres premiers p et q doivent être vraiment aléatoires, sinon il est "facile" 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 sur dans le sens où une observation d'environs 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) Donné e et $\varphi(n)$ 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 nombre 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 chaine de caractère 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 11.6 : !!! Casser le chiffrement RSA

Voici une clef publique

```
(737222068937468780393102984127683335175475064864273639134061748152408232848 57, \
339210030483975845798353604775492238287235901869178116099382740088401819684 99)
```

et un message chiffré avec cette clef publique :

```
[229738772397888738828377886878347409581450919795015651678249925978256004069 74,
 485033793619423568291279012734835806394264746008015398655258303603023506026 89,
 222479845494201229863762885581088617570424573742360886819061324916186152605 5,
 450072070121614530203662597946239741112754171159651504263530284314274804748 6,
 354450009356712804290798777475533631216457810964303864174283074852289043868 25,
 486277122595015630358064156889125604810205778057841449692453866995394638337 35,
 718683890927685890929478344411692715122278824691293667067582799607515027391 57,
 260196030194823820855057279010920921222414389591476540681174754477836025729 84,
 650397294725217069545106248289843293097122563903954161847044906833778748573 02,
 118053201413196623421355522172869278684667952806318169450323878366227984951 17]
```

a) Décrypter le message précédent !

Indication : Il est probablement nécessaire de choisir un algorithme adapté, par exemple basé sur des cribles quadratiques (QS, MPQS, SIQS).