TP 9 : Erreurs numériques, méthodes de calcul de racines.

1 Erreurs numériques

Exercice 1. *Étude d'une suite récurrente (1).* On considère la suite définie par $u_0 = 2$ et $u_{n+1} = \sin(u_n)$ pour tout $n \geq 0$. Une étude théorique montre que $u_n \underset{n\to+\infty}{\sim} \sqrt{\frac{3}{n}}$.

1. On a vu dans le cours que l'évolution de l'erreur relative dans l'évolution d'une suite donnée par $u_{n+1} = f(u_n)$ était fonction du *conditionnement* de la fonction f, dont l'expression est $\kappa(x) = \left|\frac{xf'(x)}{f(x)}\right|$. Calculer le conditionnement $\kappa(x)$ pour f la fonction sinus.

Il est facile de voir que $\kappa(x)$ est dans $[0, 1]$ pour $x \in [0, \pi/2]$. Ainsi, les valeurs de la suite que l'on calculera expérimentalement seront plutôt correctes !

2. Écrire une fonction Python `termes_suite(n)` prenant en entrée un entier $n > 0$ et renvoyant les n premiers termes de la suite, sous la forme d'une liste.

3. Vérifier expérimentalement que $u_n \underset{n\to+\infty}{\sim} \sqrt{\frac{3}{n}}$. On pourra tracer les 500 premières valeurs prises par $u_n\sqrt{\frac{n}{3}}$ en fonction de n, et vérifier que la limite est 1.

Exercice 2. *Étude d'une suite récurrente (2).* On considère la suite définie par $u_0 = 1/3$ et $u_{n+1} = 1 - 2u_n$ pour tout $n \geq 0$. Que vaut (u_n) (en mathématique) ? Calculer et afficher à l'écran les 60 premiers termes de la suite. Justifier en étudiant le conditionnement de $x \mapsto 1 - 2x$ au voisinage de 1/3.

Exercice 3. *Étude d'une suite récurrente (3).* On considère la suite $(u_n)_{n\in\mathbb{N}}$ définie par

$$u_0 = \frac{11}{2}, \qquad u_1 = \frac{61}{11} \qquad \text{et} \qquad u_n = 111 - \frac{1130 - \frac{3000}{u_{n-2}}}{u_{n-1}} \quad \text{pour tout } n \geq 2.$$

1. Calculer des approximations des 30 premiers termes de la suite à l'aide d'une liste. Quelle semble être la limite de cette suite ?

2. Il est possible de faire des calculs exacts sur les rationnels à l'aide du module `fraction`, qui s'utilise de la façon suivante (pour calculer par exemple le terme u_2 de la suite) :

```
>>> from fractions import Fraction
>>> u0=Fraction(11,2)
>>> u1=Fraction(61,11)
>>> u2=111-(1130-3000/u0)/u1
>>> print(u2)
341/61
>>> print(float(u2))
5.590163934426229
```

Modifiez votre réponse à la question précédente pour calculer les 30 premiers termes de la suite de façon plus précise. On créera également un tableau dans lequel on stockera des approximations des termes de la suite par des flottants. Par contre, ces approximations seront faites « au dernier moment » : on veillera à ce que les calculs soient effectués avec des fractions, et on stockera dans le tableau des approximations obtenues à l'aide de la fonction `float`. Quelle semble être la limite de la suite ?

3. Tracer sur une même image les graphes sur l'intervalle $[4, 105]$ des fonctions

$$x \mapsto x \qquad \text{et} \qquad f : x \mapsto 111 - \frac{1130 - \frac{3000}{x}}{x}$$

4. Tenter d'expliquer le phénomène obtenu.

2 Méthodes dichotomiques et de Newton

Dans les deux exercices suivants, on cherche à comparer méthode dichotomique et méthode de Newton pour le calcul de $\sqrt{p}$, où p est un entier que vous fixerez comme vous voulez.

Exercice 4. *Méthode dichotomique.* ***1.*** On rappelle le principe de la méthde dichotomique pour rechercher un zéro d'une fonction f continue sur un intervalle $[a, b]$, telle que $f(a)f(b) \leq 0$ (ce qui assure l'existence d'un tel zéro) : on part de l'intervalle $[x, y] = [a, b]$, et à chaque étape, on remplace x ou y par le milieu de $[x, y]$ de façon à maintenir *l'invariant de boucle* $f(x)f(y) \leq 0$. L'algorithme s'arrête lorsque $y - x$ devient plus petit qu'un petit réel fixé à l'avance. Recoder sans le cours (si possible) une fonction `zero_dichotomie(f,a,b,eps)` renvoyant un zéro à ε près d'une fonction continue sur $[a, b]$.

2. L'utiliser avec la fonction $x \mapsto x^2 - p$ sur l'intervalle $[0, p]$. Observer à l'aide d'un `print` que le nombre de chiffres corrects augmente *linéairement* avec le nombre d'étapes effectuées.

Exercice 5. *Méthode de Newton.* ***1.*** Recoder la méthode de Newton sous la forme d'une fonction `newton(f,g,x0,N)`, où :

— f est la fonction dont on cherche un zéro ;

— g est sa dérivée ;

— x_0 est le point de départ ;

— N est le nombre d'itérations effectuées.

on rappelle que pour résoudre $f(x) = 0$, on part d'un x_0 fixé, et à chaque étape, on remplace le point courant x par l'intersection de la tangente en x avec l'axe des abscisses.

2. L'utiliser avec la fonction $x \mapsto x^2 - p$, la dérivée étant $x \mapsto 2x$. On pourra partir de $x_0 = p$. Observer à l'aide d'un `print` que le nombre de chiffres corrects *double* environ à chaque étape effectuée.

Exercice 6. *Convergence lente.* Tester la méthode de Newton avec $f : x \mapsto (x - 1)^2$ et $x_0 = 2$. La convergence est-elle quadratique ? Expliquer en calculant exactement les valeurs prises, et/ou en faisant un dessin.

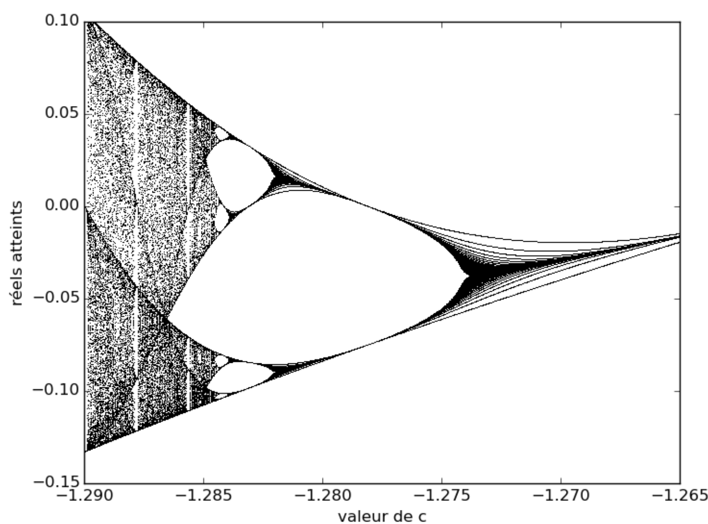


FIGURE 1: Diagramme de bifurcation (exercice 7). Axe des abscisses : le réel $c \in [-1.29, -1.265]$. Axe des ordonnées : l'intervalle $[-0.15, 0.1]$. Les points du diagramme sont les itérés de la méthode de Newton pour la résolution de $x^3 + cx + 1 = 0$, avec $x_0 = 0$, qui tombent dans l'intervalle $[-0.15, 0.1]$. Le chaos s'installe !

Exercice 7. *Approximation numérique d'un zéro d'un polynôme de degré 3.* On cherche à résoudre $x^3 + cx + 1 = 0$, avec c un réel.

1. Se convaincre qu'avec $c > 0$, la méthode de Newton converge forcément quel que soit le point de départ. On pourra tracer quelques courbes. Tester la méthode avec $c = 1$ et différents premiers termes.

2. on prend $c = 0$. Calculez les 30 premiers termes avec $x_0 = 0.8$. On peut montrer que la méthode converge (parfois difficilement) sauf pour un ensemble dénombrable de valeurs de x_0.

3. avec $c < 0$, la méthode est-elle convergente ? On essaiera par exemple $c = -1.286$ et $x_0 = 0$. Voyez ensuite la figure 1.

Exercice 8. *Utilisation de Scipy.* La méthode de Newton est implémentée dans le sous-module `optimize` du module `scipy`, l'essayer sur quelques exemples :

```
import scipy.optimize as sco
print(sco.newton(lambda x:x*x-2, 5, fprime=lambda x:2*x))
```

Remarques :

— En lisant l'aide de la fonction, est-il possible de spécifier la précision souhaitée ? Le nombre d'itérations ?

— Quelles sont les valeurs par défaut ?

3 Une introduction aux fractales

La méthode de Newton s'applique sans grande modification aux fonctions vectorielles $\mathbb{R}^k \to \mathbb{R}^k$ (la dérivée ayant un analogue). En particulier, elle s'applique sans changement aux fonctions de la variable complexe.

Pour créer le nombre complexe $x + \imath y$, on utilise simplement `complex(x,y)` (il n'y a pas besoin d'importer de module). Remarquez qu'à l'affichage, c'est la lettre `j` qui est utilisée pour la racine de $z^2 + 1$ de partie imaginaire positive, comme en physique. La fonction `abs` permet d'obtenir le module d'un complexe. Pour z un complexe, `z.real` et `z.imag` permettent d'obtenir les parties réelles et imaginaires.

Exercice 9. Appliquer la méthode de Newton (que vous avez codée !) sur la fonction $z \mapsto z^2 - K$, où K est un complexe, avec la dérivée $z \mapsto 2z$.

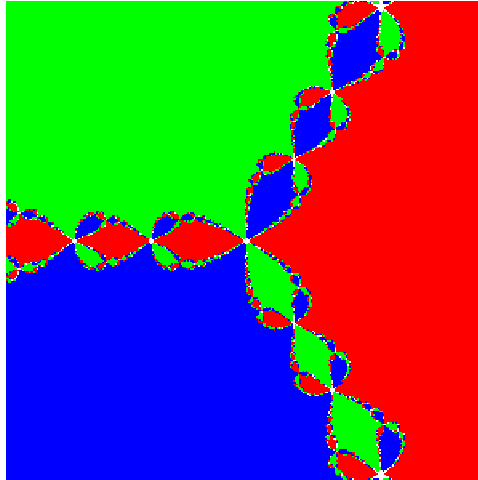
Le polynôme $z^3 - 1$ possède trois racines dans le plan complexe, à savoir $1, \jmath, \jmath^2$, où $\jmath = -\frac{1}{2} + \imath\frac{\sqrt{3}}{2}$. On va appliquer la méthode de Newton dans le plan complexe à la fonction $z \mapsto z^3 - 1$, de dérivée $z \mapsto 3z^2$.

Exercice 10. Définir `eps` et `nmax` deux variables globales (on pourra prendre par exemple `eps=0.001` et `nmax=20`). Écrire une fonction `point(z0)` prenant en paramètre un complexe, et réalisant des itérations de Newton associées à la fonction $z \mapsto z^3 - 1$, à partir de z_0. Votre fonction devra réaliser au plus `nmax` itérations, et s'arrêter si elle tombe à moins de ε d'une des racines de f. Dans ce dernier cas, elle renverra l'entier $k \in \{0, 1, 2\}$ tel que la racine approchée soit $\jmath^k$. Dans le cas contraire, elle renverra 3. Attention, il se peut que le z_0 de départ fasse que la méthode produise une erreur (une division par 0). Pour éviter les erreurs, on fera usage de `try` et `except` qui s'utilisent comme suit :

```
try:
    [instructions]
except:
    [autres instructions]
```

Python essaie d'abord de réaliser `[instructions]`, mais si une erreur se produit, il réalise `[autres instructions]`. Si une erreur se produit, votre fonction devra également renvoyer 3, ainsi votre boucle devra être dans le bloc `try`.

Exercice 11. Utiliser votre fonction pour écrire une fonction `fractale(x0,xmax,y0,ymax,Nx,Ny)` produisant une image de taille $N_y \times N_x$, telle que chaque pixel de l'image soit associé à un point du plan complexe entre $x_0 + \imath y_0$ et $x_{\max} + \imath y_{\max}$, et coloré suivant la racine atteinte (0 : point rouge, 1 : point vert, 2 : point bleu, le point sera blanc en cas de non convergence). Le résultat est donné sur la figure suivante. Voir fin du TP pour un rappel sur les images. Attention, lorsqu'on convertit un tableau en image, la première dimension est associée aux lignes, et la deuxième aux colonnes. De plus, le premier point du tableau est associé au coin en haut à gauche de l'image.



Rappels : Numpy, Matplotlib et Image

On rappelle ici des fonctions utiles provenant des modules Numpy et Matplotlib, qu'on importera comme suit :

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

Numpy

Pour convertir une liste en tableau Numpy, on utilise simplement `np.array`.

```
>>> np.array([1,2]) #vecteur (ligne)
array([1, 2])
```

`np.zeros` est utile pour créer des tableaux Numpy remplis de zéros.

```
>>> np.zeros(4)
array([ 0.,   0.,   0.,   0.])
```

`np.linspace` fournit facilement un tableau de flottants régulièrement espacés dans un intervalle : `np.linspace(a,b,N)` fournit N points dont le premier est a, le dernier b, et ils sont espacés de $\frac{b-a}{N-1}$.

```
>>> np.linspace(0,5,8)
array([ 0. ,   0.71428571,   1.42857143,   2.14285714,   2.85714286, 3.57142857,   4.28571429,   5.])
```

Matplotlib

`plt.plot(X,Y)` permet de tracer une courbe reliant par lignes brisées les points (x_i, y_i), avec `X` et `Y` deux tableaux Numpy (ou simplement deux listes) de même taille.

```
def f(x):
    return x*x+x-4

X=np.linspace(-5,5,1000)
Y=[f(x) for x in X]
plt.plot(X,Y,label="courbe")
plt.legend(loc="upper left")
#positionnement de la légende en haut à gauche. (upper/center/lower et left/right).
plt.show() #pour afficher le graphe.
```

Image

On rappelle les fonctions suivantes :

```
import numpy as np
from PIL import Image
np.linspace(x0,xmax,Nx) #crée un tableau Numpy de Nx points régulièrement espacés entre x0 et xmax.
np.zeros((a,b,3), dtype="uint8") # crée un tableau numpy à trois dimensions a x b x 3 rempli de zéros
                                 # (de type entiers naturels sur 8 bits)
image=Image.fromarray(tableau) #convertir un tableau en image.
image.show() #afficher l'image
```

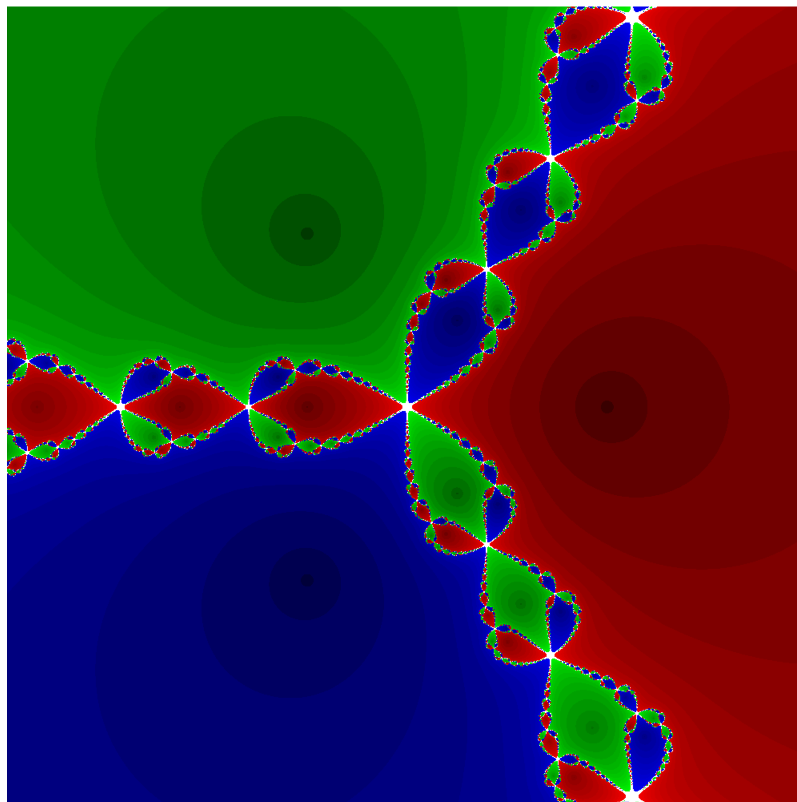


FIGURE 2: L'image obtenue entre $-2-2\imath$ et $2+2\imath$ et 10^6 points, où on fait varier la luminosité en fonction de la vitesse de convergence. Les trois points $1, \jmath$ et $\jmath^2$ sont bien visibles.