Lycée Louis-Le-Grand, Paris
MPSI 4 – Informatique commune
N. Carré, A. Troesch

2017/2018

TP n° 9 : Résolution d'équations numériques

Nous étudions dans ce TP plusieurs méthodes de calcul approché de solutions d'une équation du type $f(x) = 0$, f étant une fonction continue d'un intervalle I de $\mathbb{R}$ dans $\mathbb{R}$.

Correction de l'exercice 1 –
On commence par l'import des modules qui seront nécessaires (pour les opérations mathématiques, pour les mesures de durée, pour les résolutions numériques par les méthodes déjà implémentées, pour les tracés de graphe), et la définition des différentes fonctions qui interviendront (ainsi que leurs dérivés, afin de pouvoir utiliser la méthode de Newton) :

```
import math
import time
from scipy.optimize import newton
import matplotlib.pyplot as plt

def f1(x):
    return x*x - 2

def f2(x):
    return x - math.exp(-x)

def f3(x):
    return math.sqrt(x) - 0.5

def f4(x):
    return x**3

def f5(x):
    return x - 1 / 4 * (x** 3 - x)

def df1(x):
    return 2 * x

def df2(x):
    return 1 + math.exp(-x)

def df3(x):
    return 1 / (2 *math.sqrt(x))

def df4(x):
    return 3 * x * x

def df5(x):
    return 1 - 1 / 4 * (3* x * x - 1)
```

1. **Dichotomie**

   On rappelle que la méthode de dichotomie consiste à partir d'un intervalle $[a, b]$ tel que $f(a)$ et $f(b)$ soient de signes opposés. On itère le procédé consistant à couper l'intervalle en son milieu, et à garder la moitié assurant un changement de signe, ceci jusqu'à ce qu'on obtienne un intervalle de longueur suffisamment petite.

   On commence par s'assurer que la méthode de dichotomie est applicable dans cette situation, c'est-à-dire que $f(a)$ et $f(b)$ n'ont pas même signe.

   Attention, si le test est valide, cela nous assure de l'existence de la solution (en supposant f continue), ainsi que de la convergence de la méthode. En revanche, le fait que le test soit non valide n'implique pas la non existence d'un zéro. Si $f(a)$ et $f(b)$ sont tous les deux positifs, f peut tout de même prendre des valeurs négatives entre les deux. Il convient donc de na pas renvoyer dans ce cas un message d'erreur stipulant qu'il n'existe pas de zéros dans l'intervalle !

   Enfin, concernant la condition d'arrêt, puisqu'à la fin, on retourne le milieu du dernier intervalle obtenu, il suffit que cet intervalle soit de rayon au plus ϵ, donc de diamètre au plus 2ϵ (c'est ce diamètre qu'on compare, en effectuant le test $b - a < \varepsilon$).

   On obtient le code suivant :

```
def dichotomie(f,a,b,epsilon):
    if f(a)*f(b) > 0:
        raise ValueError('f(a) et f(b) sont de même signe')
    if b<a:
        a,b = b,a
    epsilon2 = 2 * epsilon
    k = 0
    while b-a > epsilon2:
        c = (a+b)/2
        if f(a)*f(c) <= 0:
            b = c
        else:
            a = c
        k += 1
    return ((a+b)/2, k)
```

   Nous effectuerons les tests de validité globalement, pour toutes les méthodes réunies, plus tard.

2. **Méthode de la fausse position**

   On rappelle que la méthode de la fausse position (ou *regula falsi*, ou méthode d'interpolation linéaire) consiste à partir d'un intervalle $[a, b]$ tel que $f(a)$ et $f(b)$ soient de même signe, puis à itérer le procédé consistant à couper l'intervalle au point c en lequel la corde issue des abscisses a et b (c'est-à-dire la courbe interpolatrice aux points a et b, consistant en une interpolation linéaire) rencontre l'axe des abscisses, puis à garder, des deux intervalles obtenus, celui qui assure un changement de signe.

   On a vu dans le cours que si f est continue, l'une des deux bornes converge vers un zéro de f. Cependant, en général, l'autre borne ne converge pas vers ce zéro, donc la longueur de l'intervalle ne peut pas être utilisée comme condition d'arrêt. On décide de s'arrêter lorsque $f(c - \varepsilon)$ et $f(c + \varepsilon)$ ne sont pas de même signe, ce qui assure l'existence d'un zéro dans l'intervalle $]c - \varepsilon, c + \varepsilon[$.

```
def regulafalsi(f,a,b,epsilon):
    if f(a)*f(b) > 0:
        raise ValueError('f(a) et f(b) sont de même signe')
    if b<a:
        a,b = b,a
    a0,b0=a,b
    k = 0
    if abs(f(a)) < 1e-15: # comparaison à 0, en gérant l'inexactitude
```

```
        return(a)
    if abs(f(b)) < 1e-15:
        return(b)
    c = a - (b-a)* f(a) / (f(b)-f(a)) # dans les cas restants f(a) != f(b)
    while f(max(a0,c-epsilon))*f(min(b0,c+epsilon))>0:
        if f(a)*f(c) < 0:
            b = c
        else:
            a = c
        c = a - (b-a)* f(a) / (f(b)-f(a))
        k += 1
    return (c,k)
```

Encore une fois, on remet les tests (et notamment le cas de la fonction $x \mapsto x^3$) à plus tard.

3. **Méthode de la sécante** Le principe est le même que pour la méthode de fausse position, mais au lieu conserver l'intervalle assurant l'existence d'un zéro, on conserve systématiquement l'intervalle dont bornes sont les deux dernières valeurs calculées. Ainsi, si les c_k sont les différentes valeurs calculées aux différentes étapes (en posant $a = c_0$, $b = c_1$ et c_2 la valeur de la première intersection calculée), on considère systématiquement les intervalles $[c_k, c_{k+1}]$, même si l'existence d'un zéro dans l'intervalle n'est pas assurée. Dans ce cas, la valeur de c_{k+2} ne sera pas dans l'intervalle (et on peut espérer récupérer le zéro)

   Nous avons vu dans le cours qu'en général (si $f'(z) \neq 0$), à condition d'initialiser pour des valeurs suffisamment proches du zéro z, on aura convergence de la méthode, et ceci assez rapidement (la méthode étant d'ordre φ le nombre d'or, c'est-à-dire que la convergence est contrôlée par un terme de l'ordre de a^{φ^n}, où a est un réel de $]0, 1[$).

   Cette convergence rapide nous incite à décréter l'échec de la méthode (donc sa non convergence) au bout d'un nombre (qu'on n'est pas obligé de choisir trop grand) d'itérations, si on n'a pas obtenu de convergence à ce moment. On décrète donc que si au bout de 100 itérations, on n'observe pas de convergence, on peut s'arrêter. On aurait pu se contenter d'aller jusqu'à 50 itérations (c'est le cas de la méthode `newton` implémentée dans le module `scipy.optimize`).

   La condition d'arrêt qu'on suggère est l'obtention de deux valeurs consécutives suffisamment proches l'une de l'autre. La validité de cette condition d'arrêt provient du fait que si on est dans les conditions du théorème de convergence de la méthode, la convergence est très rapide, et si deux termes sont ε-proches l'un de l'autre, les suivants seront encore beaucoup plus proches les uns des autres, de sorte que l'erreur accumulée sur la suite du calcul sera inférieure à ε.

```
def secante(f,a,b,epsilon):
    if f(a) == f(b):
        if abs(f(a)) > 1e-15:
            raise ValueError('méthode non définie pour cette initialisation')
        else:
            return (a,0)
    a,b = b, a - (b-a)* f(a) / (f(b)-f(a))
    # on fait une première itération pour éviter le problème d'une
    # initialisation par deux valeurs trop rapprochées et non
    # significatives
    k = 0
    while (abs(b-a) >= epsilon) and (k<100):
        a,b = b, a - (b-a)* f(a) / (f(b)-f(a))
        k += 1
    if k<100:
        return (b,k)
    else:
```

```
        raise ValueError('méthode divergente')
```

4. **Méthode de Newton-Raphson** Enfin, la méthode de Newton-Raphson est en quelque sorte un passage à la limite sur la méthode de la sécante : si deux points sont proches, la corde en ces deux points est à peu près égale à la tangente en l'un de ces deux points. Ainsi, la méthode de Newton consiste à remplacer dans la méthode de la sécante la corde par la tangente à la dernière borne calculée de l'intervalle. En particulier, la borne précédente n'est plus utilisée dans le calcul, et la donnée de l'intervalle n'est plus pertinent (sauf pour contrôler sa longueur, donc pour décider de la condition d'arrêt)

   Comme pour la méthode de la sécante, sauf dans le cas $f'(z) = 0$, la méthode est convergente dès lors que l'initialisation se fait suffisamment proche du zéro z recherché, et dans ce cas, la convergence est extrêmement rapide d'ordre 2 (donc de l'ordre de a^{2^n}, pour $a \in]0, 1[$). Ceci donne la validité d'une condition d'arrêt du même type que pour la méthode de la sécante : si deux valeurs consécutives sont proches, les autres le sont encore tellement plus que l'erreur accumulée ensuite est petite.

   L'inconvénient de cette méthode est qu'elle nécessite la donnée de la dérivée.

```
def newtonraphson(f,df,a,epsilon):
    b = a - f(a) / df(a)
    k = 0
    while (abs(b-a) >= epsilon) and (k<100):
        a,b = b, b - f(b) / df(b)
        k +=1
    if k < 100:
        return (b,k)
    else:
        raise ValueError('méthode divergente')
```

   On effectue maintenant les tests comparatifs. Pour commencer, on assure la validité des résultats en comarant les valeurs obtenues par les fonctions qu'on vient de programmer, ainsi que les valeurs théoriques, et les valeurs calculées par la méthode de Newton déjà implémentée (la fonction `newton` prenant de façon optionnelle la dérivée f' en paramètre, elle consiste soit en la méthode de Newton si la dérivée est fournie, soit en la méthode de la sécante, si la dérivée n'est pas fournie).

```
def comparevaleurs(f,df,a,b,epsilon,valexacte):
    print('valeur exacte: {}'.format(valexacte))
    print('Par scipy.optimize.newton avec dérivée: {}'.format(newton(f,b,df)))
    print('Par scipy.optimize.newton sans dérivée: {}'.format(newton(f,b)))
    print('Par dichotomie: {}'.format(dichotomie(f,a,b,epsilon)[0]))
    print('Par fausse position: {}'.format(regulafalsi(f,a,b,epsilon)[0]))
    print('Par sécante: {}'.format(secante(f,a,b,epsilon)[0]))
    print('Par newton: {}'.format(newtonraphson(f,df,b,epsilon)[0]))
    print()

print('Résultats pour la fonction f1:')
comparevaleurs(f1,df1,0,2,1e-10,math.sqrt(2))

print('Résultats pour la fonction f2:')
comparevaleurs(f2,df2,0,2,1e-10,'non connue')

print('Résultats pour la fonction f3:')
comparevaleurs(f3,df3,0,0.5,1e-10,0.25)
```

   On obtient les résultats suivants :

```
Résultats pour la fonction f1:
valeur exacte: 1.4142135623730951
Par scipy.optimize.newton avec dérivée: 1.4142135623730951
Par scipy.optimize.newton sans dérivée: 1.4142135623730954
Par dichotomie: 1.4142135623260401
Par fausse position: 1.4142135623189167
Par sécante: 1.414213562373095
Par newton: 1.4142135623730951

Résultats pour la fonction f2:
valeur exacte: non connue
Par scipy.optimize.newton avec dérivée: 0.5671432904097838
Par scipy.optimize.newton sans dérivée: 0.5671432904097952
Par dichotomie: 0.5671432903618552
Par fausse position: 0.5671432904470005
Par sécante: 0.5671432904097838
Par newton: 0.5671432904097838

Résultats pour la fonction f3:
valeur exacte: 0.25
Par scipy.optimize.newton avec dérivée: 0.25
Par scipy.optimize.newton sans dérivée: 0.25000000000000605
Par dichotomie: 0.24999999994179234
Par fausse position: 0.25000000008069295
Par sécante: 0.25000000000000006
Par newton: 0.25
```

Les méthodes semblent correctes, et on constate que les conditions d'arrêt pour la sécante et Newton donnent des résultats cohérents.

On compare maintenant le nombre d'itérations nécessaires pour chacune de ces méthodes :

```
def nombre_iterations(f,df,a,b,epsilon):
    print('Par dichotomie: {}'.format(dichotomie(f,a,b,epsilon)[1]))
    print('Par fausse position: {}'.format(regulafalsi(f,a,b,epsilon)[1]))
    print('Par sécante: {}'.format(secante(f,a,b,epsilon)[1]))
    print('Par newton: {}'.format(newtonraphson(f,df,b,epsilon)[1]))
    print()

print("Nombres d'itérations pour la fonction f1:")
nombre_iterations(f1,df1,0,2,1e-10)

print("Nombres d'itérations pour la fonction f2:")
nombre_iterations(f2,df2,0,2,1e-10)

print("Nombres d'itérations pour la fonction f3:")
nombre_iterations(f3,df3,0,0.5,1e-10)
```

Les résultats obtenus sont :

```
Nombres d'itérations pour la fonction f1:
Par dichotomie: 34
Par fausse position: 13
Par sécante: 7
```

```
Par newton: 4

Nombres d'itérations pour la fonction f2:
Par dichotomie: 34
Par fausse position: 10
Par sécante: 6
Par newton: 4

Nombres d'itérations pour la fonction f3:
Par dichotomie: 32
Par fausse position: 30
Par sécante: 6
Par newton: 4
```

On peut se rendre compte de la convergence extrêmement rapide des deux dernières méthodes (avec un petit avantage pour le méthode de Newton) La méthode de la fausse position est sur ces exemples meilleure que la dichotomie. Les trois méthodes assurent ici une convergence raisonnable.

On compare les durées d'exécution :

```
def duree_execution(f,df,a,b,epsilon):
    debut = time.time()
    dichotomie(f,a,b,epsilon)
    fin = time.time()
    print('Par dichotomie: {}'.format(fin - debut))
    debut = time.time()
    regulafalsi(f,a,b,epsilon)
    fin = time.time()
    print('Par fausse position: {}'.format(fin - debut))
    debut = time.time()
    secante(f,a,b,epsilon)
    fin = time.time()
    print('Par sécante: {}'.format(fin - debut))
    debut = time.time()
    newtonraphson(f,df,b,epsilon)
    fin = time.time()
    print('Par newton: {}'.format(fin - debut))
    print()

print("Durée d'exécution pour la fonction f1:")
duree_execution(f1,df1,0,2,1e-10)

print("Durée d'exécution pour la fonction f2:")
duree_execution(f2,df2,0,2,1e-10)

print("Durée d'exécution pour la fonction f3:")
duree_execution(f3,df3,0,0.5,1e-10)
```

Les résultats obtenus :

```
Durée d'exécution pour la fonction f1:
Par dichotomie: 3.0994415283203125e-05
Par fausse position: 4.220008850097656e-05
Par sécante: 1.2636184692382812e-05
```

```
Par newton: 6.4373016357421875e-06

Durée d'exécution pour la fonction f2:
Par dichotomie: 4.029273986816406e-05
Par fausse position: 4.7206878662109375e-05
Par sécante: 1.4781951904296875e-05
Par newton: 8.821487426757812e-06

Durée d'exécution pour la fonction f3:
Par dichotomie: 3.337860107421875e-05
Par fausse position: 0.00010371208190917969
Par sécante: 1.239776611328125e-05
Par newton: 8.58306884765625e-06
```

On se rend compte que la simplicité de la dichotomie (des calculs simples à chaque itération et une condition d'arrêt plus simple à vérifier que pour la méthode de la sécante) lui fait gagner du temps par rapport à la méthode de la fausse position, et pour la marge d'erreur donnée, donne des temps dont l'ordre de grandeur n'est pas démesuré par rapport aux méthodes de la sécante et de Newton.

Pour que ces deux dernières méthodes révèlent toute leur efficacité, il faudrait travailler avec des précisions plus élevées (donc pas en norme IEEE 754). Par exemple, pour obtenir 1000 décimales, la méthode de Newton doublant le nombre de décimales correctes à chaque itération, avec une initialisation à 1 près, on a l'approximation recherchée en une dizaine d'itérations, alors que par dichotomie, il en faudra plus de 3000. Sachant que lorsqu'on travaille avec un grand nombre de décimales, chaque opération élémentaire est elle même assez longue, un tel gain sur le nombre d'itérations est appréciable !

Nous étudions maintenant le cas de la fonction $f_4 : x \mapsto x^3$, qui présente une tangente horizontale en $z = 0$, et nous initialisons les méthodes sur l'intervalle $[-\frac{1}{2}, 1]$. Dans cette situation les convergences sont moins rapide (à cause de la platitude de la fonction). On se contente d'une erreur de 10^{-3}.

```
print('Valeurs pour la fonction f4 (erreur 1e-3):')
print('Par dichotomie: {}'.format(dichotomie(f4,-0.5,1,1e-3)[0]))
print('Par fausse position: {}'.format(regulafalsi(f4,-0.5,1,1e-3)[0]))
print('Par sécante: {}'.format(secante(f4,-0.5,1,1e-3)[0]))
print('Par newton: {}'.format(newtonraphson(f4,df4,1,1e-3)[0]))
print()

print("Nombre d'itérations pour f4 (erreur 1e-3):")
nombre_iterations(f4,df4,-0.5,1,1e-3)

print("Durée d'exécution pour la fonction f4 (erreur 1e-3):")
duree_execution(f4,df4,-0.5,1,1e-3)
```

Les résultats obtenus sont les suivants :

```
Valeurs pour la fonction f4 (erreur 1e-3):
Par dichotomie: 0.000244140625
Par fausse position: -0.000999999357696357
Par sécante: -0.0030578316181607756
Par newton: 0.0015224388403474454

Nombre d'itérations pour f4 (erreur 1e-3):
Par dichotomie: 10
Par fausse position: 498996
Par sécante: 17
```

```
Par␣newton:␣15

Durée␣d'exécution pour la fonction f4 (erreur 1e-3):
Par dichotomie: 1.5497207641601562e-05
Par fausse position: 1.7400398254394531
Par sécante: 4.2438507080078125e-05
Par newton: 2.193450927734375e-05
```

On se rend compte que l'hypothèse $f'(z) = 0$ diminue considérablement l'efficacité des différentes méthodes (sauf la méthode de dichotomie dont la vitesse de convergence ne dépend pas de f) Si les méthodes de la sécante et de Newton restent raisonnables, il n'en est pas de même de la méthode de la fausse position. On peut en effet montrer que pour cet exemple, la convergence est de l'ordre de $\frac{1}{\sqrt{n}}$. Si on essaie un calcul à 10^{-4} près, le temps de calcul devient vraiment très long !)

Il faut donc retenir que si la méthode de Newton (ou de la sécante) est la plus efficace dans de nombreuses situations, il existe des situations où la méthode de dichotomie est meilleure, cette méthode étant beaucoup plus stable (indépendante du comportement de la fonction).

Nous poussons la précision un peu plus loin sur le même exemple, mais en supprimant la méthode de la fausse position :

```
print("Valeurs␣et␣nombre␣d'itéations␣pour␣la␣fonction␣f4␣(erreur␣1e-13):")
print('Par␣dichotomie:␣{}'.format(dichotomie(f4,-0.5,1,1e-13)))
print('Par␣sécante:␣{}'.format(secante(f4,-0.5,1,1e-13)))
print('Par␣newton:␣{}'.format(newtonraphson(f4,df4,1,1e-13)))
```

Nous obtenons cette fois :

```
Valeurs et nombre d'itéations␣pour␣la␣fonction␣f4␣(erreur␣1e-13):
Par␣dichotomie:␣(-2.842170943040401e-14,␣43)
Par␣sécante:␣(-2.9600077014861837e-13,␣99)
Par␣newton:␣(1.3974558270919538e-13,␣72)
```

Remarquons au passage que dans cette situation où la convergence est beaucoup plus lente, la condition d'arrêt donnée pour les deux dernières méthodes est discutable (les valeurs données sont des approximations insuffiantes du résultat attendu). Par ailleurs, si on avait demandé une approximation légèrement plus précise, on nous aurait retourné une divergence de la méthode de la sécante (plus de 100 itérations).

Enfin, la fonction f_5 est un exemple de fonction telle que la méthode de Newton soit définie mais divergente, alors que la fonction admet un zéro. L'idée de cet exemple est de construire une fonction f telle que $f(1) = 1$ et la dérivée en 1 soit de pente $\frac{1}{2}$, donc la tangente recoupe l'axe des abscisses en -1. On impose alors que $f(-1) = -1$ et la dérivée en -1 soit aussi de pente $\frac{1}{2}$, de sorte que la tangente recoupe l'axe des abscisses en 1. Ainsi, la méthode de Newton initialisée en 1 boucle périodiquement sur les deux valeurs 1 et -1.

Pour construire une telle fonction, en imposant de plus $f(0) = 0$, on considère le polynôme interpolateur de Lagrange aux points -1, 0 et 1, qui n'est autre que $P(X) = X$, et on ajoute à ce polynôme un polynôme s'annulant en -1, 0 et 1, de sorte à obtenir les dérivées souhaitées. Ainsi, on considère

$$Q(X) = X + \lambda X(X^2 - 1).$$

Un petit calcul de dérivée détermine la valeur de λ pour obtenie les conditions souhaitées sur f'.

On fait l'essai : l'instruction `newtonraphson(f5,df5,1,1e-2)` retourne le message suivant :

```
Traceback (most recent call last):
  File "py040.py", line 207, in <module>
    newtonraphson(f5,df5,1,1e-2)
  File "py040.py", line 102, in newtonraphson
    raise ValueError('méthode␣divergente')
```

```
ValueError: méthode divergente
```

5. **Dérivation numérique**

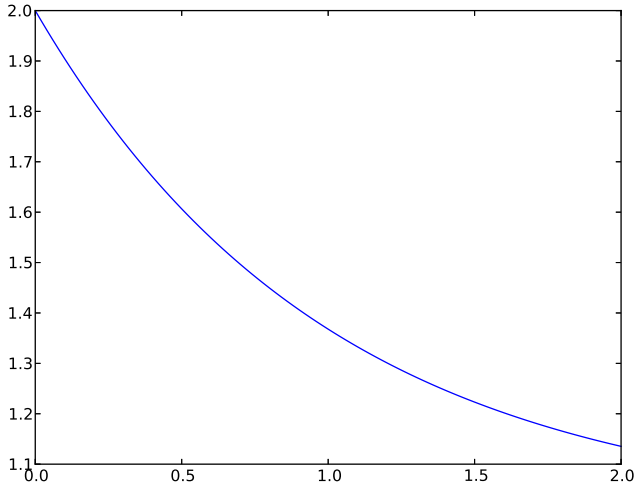
Nous approchons la dérivée de f par le taux d'accroissement $\frac{f(x+h)-f(x-h)}{2h}$. Pour une fonction donnée dont nous connaissons la dérivée théorique (nous avons pris la fonction f_2 au point 1), on compare la dérivée théorique en un point et la dérivée numérique obtenue, en faisant varier h, de sorte à déterminer quel est le choix optimal de h.

```
def derivee(f,x,h):
    return (f(x+h)-f(x-h)) / (2* h)

def trace_derivee(f,a,b,h):
    abscisses = [ a + h + i * (b - a - 2*h)/100 for i in range(101)]
    ordonnees = [ derivee(f,x,h) for x in abscisses]
    plt.plot(abscisses,ordonnees)
    plt.show()

def choix_de_h(f,df,x):
    abscisses = [y/100 for y in range(400,751)]
    ordonnees = [abs(derivee(f,x,10**(-y)) - df(x)) for y in abscisses]
    plt.plot(abscisses,ordonnees)
    plt.show()
```

Pour la fonction f_2, nous obtenons le graphe suivant pour la dérivée :



Pour l'expression de l'erreur faite en fonction de $h = 10^{-y}$, cela nous donne le graphe de la figure 1.

Ce graphe confirme le choix optimal de $h = 10^{-5}$ à 10^{-6}, pour une dérivée de l'ordre de grandeur de 1 (annoncé dans le cours).

Ceci s'explique bien par le fait qu'en appliquant la formule de Taylor-Young entre x et $x+h$ et entre x et $x-h$ à l'ordre 3, et en faisant la différence puis en quotientant par $2h$, on obtient :

$$\frac{f(x+h)-f(x-h)}{2h} - f'(x) = O(h^2).$$

L'erreur théorique sera donc de l'ordre de αh^2. Par ailleurs, les calculs fonc intervenir des grandeurs de l'ordre de $\frac{f(x)}{h}$, donc, avec l'hypothèse qu'on s'est donnée, de l'ordre de $\frac{1}{h}$. Ainsi, si r désigne l'erreur élémentaire absolue (en gros 10^{-16} à 10^{-17}, cela représente l'ordre de grandeur de la plus petite chose qu'on peut écrire avec le nombre de décimales dont on dispose, et sans utiliser d'exposants), les calculs se feront avec une approximation

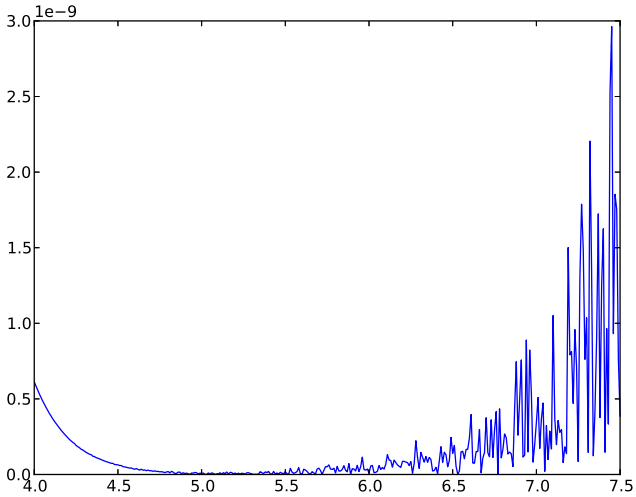


FIGURE 1 – Graphe de l'erreur pour la dérivation numérique en fonction du pas $h = 10^{-y}$

de l'ordre de $\frac{r}{h}$. Ainsi, l'erreur d'approximation est de l'ordre de $\frac{r}{h}$. L'erreur totale est donc de l'ordre de

$$f(h) = \alpha h^2 + \frac{r}{h}.$$

On minimise l'erreur en cherchant les zéros de la dérivée de f :

$$f'(h) = 2\alpha h - \frac{r}{h^2},$$

et on se rend compte que l'erreur est minimale pour $h = \sqrt[3]{\frac{r}{2\alpha}}$. Pour un point générique, on peut faire la supposition que 2α est de l'ordre de grandeur de 1, il nous reste alors la valeur optimale $h = \sqrt[3]{r}$. C'est cohérent avec les résultats pratiques trouvés.

Évidemment, le calcul effectué dépend beaucoup de l'ordre de grandeur de f et de ses dérivées (pour contrôler α), et ce résultat n'a donc pas nécessairement une validité universelle. On peut trouver des valeurs de h optimales un peu plus petites ou un peu plus grandes suivant les fonctions et le point en lequel on les évalue.

On reprogramme la fonction de Newton de sorte à ne demander que la fonction f, la fonction f' étant calculée numériquement :

```
def newtonbis(f,a,epsilon):
    return newtonraphson(f, lambda x: derivee(f,x,1e-6), a, epsilon)
```

Par exemple, pour la fonction f_1, initialisée en 2 et avec une erreur de 10^{-10}, on obtient la valeur 1.4142135623730951 au bout de 4 itérations. Le fait d'utiliser une dérivation numérique ne semble pas trop gênant pour l'efficacité de la méthode.

Correction de l'exercice 2 – Pour pouvoir utiliser la méthode de Newton, on calcule la dérivée du polynôme P :

$$P'(X) = 3X^2 - \left(\frac{3}{4} + a^2\right).$$

L'entête du programme consiste alors en l'import des modules complémentaires qui nous seront utiles (`numpy` pour créer un tableau nul sans effort, `math` pour les opérations mathématiques, et `maplotlib.pyplot` pour les représentations graphiques), ainsi que la définition des fonctions.

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

```
def f(x,a):
    return (x-1) * (x+ 0.5 - a) * (x + 0.5 + a)
def df(x,a):
    return 3 * x**2 - (0.75 + a**2)
```

On implémente ensuite la méthode de Newton initialisée en un point complexe z. Le réel a est le paramètre du polynôme P. Dans un premier temps, il peut être intéressant de travailler avec une précision plus petite, et une condition d'arrêt pour cause de divergence moins élevée, le temps que le programme soit mis en place correctement (du fait du grand nombre de répétition de ces calculs que nous allons faire, afin de pouvoir faire des essais plus rapidement). Une précision plus grande peut être ensuite exigée pour la version finale.

```
def iteration(z,a):
    t = z - f(z,a) / df(z,a)
    k = 0
    while (abs(z - t)> 1e-10) and (k < 100):
        z,t = t, t - f(t,a) / df(t,a)
        k += 1
    return (t,k)
```

On remplit ensuite le tableau des valeurs de z : chaque case su tableau représente un élément du carré plan complexe défini par $u \leqslant \Re(z) \leqslant v$ et $u \leqslant \mathrm{Im}(z) \leqslant v$, qu'on a pixélisé. Ainsi, la case (j,k) du tableau représente la donnée initiale

$$z = (u + jp) = \mathrm{i}(u + kp) \text{ où } p = \frac{v-u}{n}.$$

On rappelle que `1j` désigne en Python le nombre complexe i. Ainsi, l'itération pour chacun des complexes représentés par une case du tableau se fait de la façon suivant :

```
def tableau(a,u,v,n):
    p = (v-u) / n
    tab = np.zeros((n+1,n+1))
    for i in range(n+1):
        for j in range(n+1):
            z = u + i * p + 1j * (u + j * p)
            t,k = iteration(z,a)
            #print(t)
            if k < 100:
                if abs(t - 1) < 1e-4:
                    tab[i,j] = 1
                elif abs(t + 0.5 - a) < 1e-4:
                    tab[i,j] = 2
                else:
                    tab[i,j] = 3
    return tab
```

On attribue une valeur de 0 à 4 aux cases du tableau suivant que la méthode est divergente (valeur 0) ou convergente vers l'une des trois racines (on compare la valeur obtenue à chacun de ces racines qu'on connait, afin d'attribuer la bonne valeur). En théorie, il faudrait aussi tenir compte de la possibilité que la méthode ne soit pas définie (si la dérivée s'annule en un point obtenu). Il faudrait alors mettre une structure de gestion d'exception, ce qui ralentirait les calculs. On se rend compte que dans les cas étudiés, on ne tombe pas sur cette situation. Nous nous dispensons donc de cette structure.
Voici les résultats obtenus :

- Fractale de Newton :

```
tab = tableau(1j * math.sqrt(3) / 2, -2,2,500)
```

```
plt.matshow(tab)
plt.show()
```

La fractale obtenue est celle de la figure 2

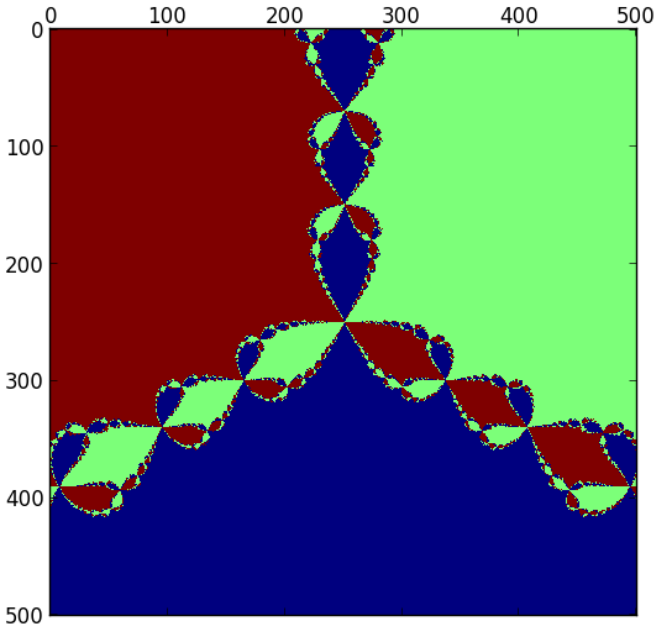


FIGURE 2 – Fractale de Newton

- Pour obtenir un lapin de Douady, on utilise les paramètres fournis (d'autres valeurs de a peuvent fournir d'autres lapins assez ressemblants, vous pouvez faire l'essai)

```
tab = tableau(-0.00508+1j * 0.33136, -2,2,500)
plt.show()
```

On entr'aperçoit sur la fractale obtenue (figure 3) cette figure un minuscule lapin de Douady bleu au centre (et sur les autres zones similaires). En faisant un zoom (c'est-à-dire en prenant $u = -0.1$ et $v = 0.1$), on a un meilleur aperçu du lapin (figure 4).
Ce lapin fractal représente une portion de la zone de divergence.

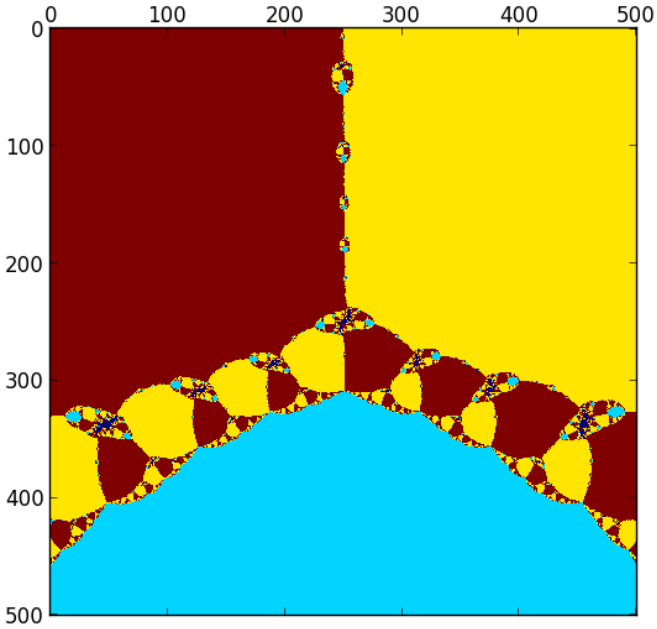


FIGURE 3 – Petit lapin de Douady

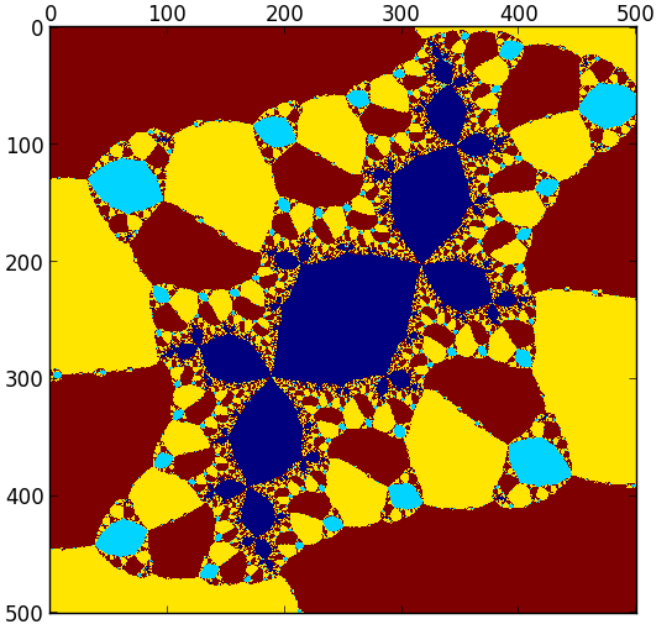


FIGURE 4 – Zoom sur le lapin de Douady