Corrigé

Méthode de Newton

Exercice 1. Méthodes de la sécante et de Newton-Raphson

On définit les fonctions suivantes :

```
def newton(f, df, u, xtol=1e-12, Nmax=100):
    n = 1
    v = u - f(u) / df(u)
    while abs (u - v) >= xtol:
        u, v = v, v - f(v) / df(v)
        n += 1
        if n > Nmax:
            return None
    return v, n
```

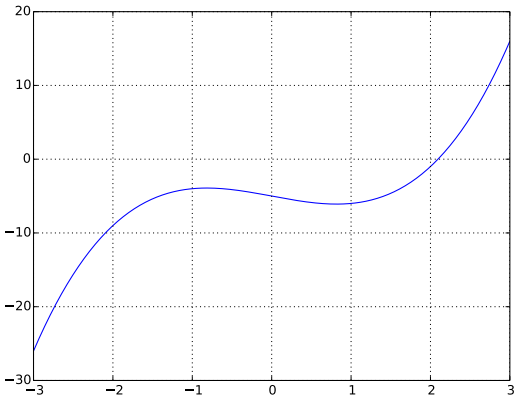
```
def secante(f, u, v, xtol=1e-12, Nmax=100):
    n = 1
    while abs (u - v) >= xtol:
        u, v = v, (u * f(v) - v * f(u)) / (f(v) - f(u))
        n += 1
        if n > Nmax:
            return None
    return v, n
```

Considérons la fonction $f : x \mapsto x^3 - 2x - 5$, et commençons par observer son tracé à l'aide du script :

```
def f(x):
    return x**3 - 2 * x - 5

def df(x):
    return 3 * x * x - 2

X = np.linspace(-3, 3, 256)
Y = [f(x) for x in X]
plt.plot(X, Y)
plt.grid()
plt.show()
```



Les deux zéros de sa dérivées f' sont dans l'intervalle $[-3,3]$ donc f est strictement monotone (ici croissante) hors de cet intervalle. f ne peut donc avoir d'autre zéro que celui qui est au voisinage de 2.
Essayons d'appliquer la méthode de Newton-Raphson à partir d'une valeur $u_0 \in [\![-3,3]\!]$. Pour cela, on applique le script :

```
for u in range(-3, 4):
    x, n = newton(f, df, u)
    print('Pour u0 = {} on obtient {} au bout de {} itérations'.format(u, x, n))
```

Voici ce qu'on obtient :

```
Pour u0 = -3 on obtient 2.0945514815423265 au bout de 8 itérations
Pour u0 = -2 on obtient 2.0945514815423265 au bout de 9 itérations
Pour u0 = -1 on obtient 2.0945514815423265 au bout de 7 itérations
Pour u0 = 0 on obtient 2.0945514815423265 au bout de 20 itérations
Pour u0 = 1 on obtient 2.0945514815423265 au bout de 10 itérations
Pour u0 = 2 on obtient 2.0945514815423265 au bout de 5 itérations
Pour u0 = 3 on obtient 2.0945514815423265 au bout de 6 itérations
```

Dans le pire des cas, il faut 20 itérations pour obtenir la précision de 10^{-12} souhaitée. On peut noter que comme on pouvait s'y attendre, c'est en partant de 2, l'entier le plus proche du zéro de f, que la méthode est la plus rapide.
Pour la méthode de la sécante, on cherche le plus mauvais des couples (u_0, u_1) à l'aide du script :

```
nmax = 0
for u in range(-3, 4):
    for v in range(-3, 4):
        if u != v:
            x, n = secante(f, u, v)
            if n > nmax:
                nmax = n
                (a, b) = (u, v)
print('pour u0 = {} et u1 = {}, {} itérations sont nécessaires'.format(a, b, nmax))
```

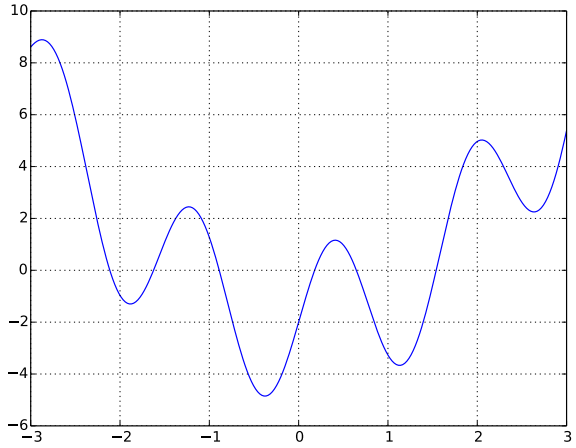
On obtient que pour $u_0 = -2$ et $u_1 = -3$, il a fallu 27 itérations pour obtenir la précision souhaitée.
Pour obtenir une situation d'échec, essayons de partir au voisinage d'une racine de f' de sorte que u_1 soit très éloigné. Pour cela on utilise l'une des deux méthodes pour en calculer une valeur approchée :

```
>>> secante(df, 0, 1)[0]
0.816496580927726
```

Et en effet, la méthode de Newton-Raphson à partir de $a = 0{,}816\,496\,580\,927\,726$ ne retourne aucun résultat. Cependant, en augmentant le nombre maximal d'itérations on finit par trouver la racine de f :

```
>>> newton(f, df, 0.816496580927726, Nmax=200)
(2.0945514815423265, 122)
```

Considérons maintenant la fonction $f : x \mapsto 3\sin(4x) + x^2 - 2$, ainsi que son graphe sur $[-3,3]$:



Cette fonction possède six zéros sur $[-3,3]$ et ne peut en posséder d'autres car $|x|>3 \Rightarrow x^2-2>7$ alors que $|3\sin(4x)| \leqslant 3$.
On définit alors la fonction suivante, pour calculer tous les zéros de f :

```
from numpy.random import random

def zeros(f, df, a, b, n=100):
    z = []
    for k in range(n):
        u = a + random() * (b - a)
        x = newton(f, df, u)
        if x is not None and round(x[0], 12) not in z:
            z.append(round(x[0], 12))
    return z
```

Elle fournit bien six résultats, approximations des différents zéros de f :

```
>>> zeros(f, df, -3, 3)
[-2.1143251763030002, -1.6244411363689999, -0.88913607483299995, 0.178877027671,
0.64653403135599996, 1.5398013104649999]
```

Exercice 2. Tache d'Airy

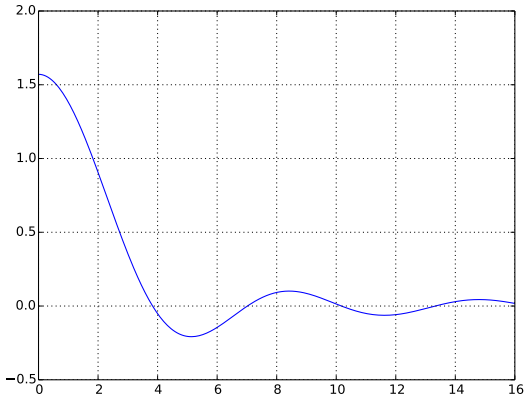
Commençons par définir la fonction f ainsi que sa dérivée :

```
from scipy.integrate import quad

def f(x):
    return quad(lambda t: np.sqrt(1-t*t)*np.cos(x*t), -1, 1)[0]

def df(x):
    return quad(lambda t: -t*np.sqrt(1-t*t)*np.sin(x*t), -1, 1)[0]
```

Le tracé du graphe de f sur l'intervalle $[0,16]$ montre que les trois premiers zéros de f sont voisins de 4, 7, 10 :



On obtient leurs valeurs respectives à l'aide de la fonction `newton` :

```
>>> newton(f, df, 4)
(3.8317059702063574, 5)
>>> newton(f, df, 7)
(7.015586669815758, 4)
>>> newton(f, df, 10)
(10.173468135063986, 4)
```

Exercice 3. Dérivation numérique

On calcule la dérivée numérique d'une fonction en un point à l'aide de la fonction :

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

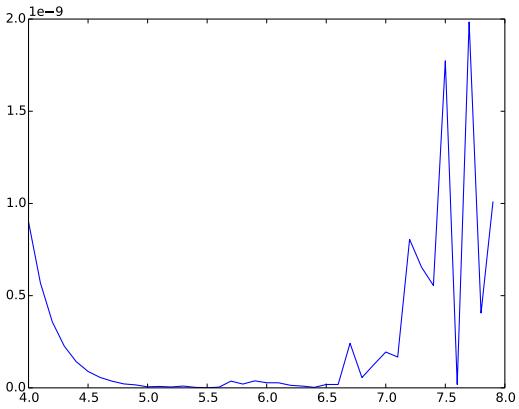
Définissons maintenant la fonction $p \mapsto \delta(1, 10^{-p})$ pour la fonction sin :

```
def delta(p):
    return abs(np.cos(1) - derivee(np.sin, 1, 10**(-p)))
```

et traçons son graphe entre 4 et 8 :

```
P = np.arange(4, 8, .1)
D = [delta(p) for p in P]

plt.plot(P, D)
plt.show()
```



Le graphe est parlant : avant la valeur optimale $p = 5$ ce sont les erreurs de nature mathématique qui prédominent, après cette valeur ce sont les incertitudes liées au calcul numérique. Ces résultats sont bien conformes à l'étude théorique qui a été faite en cours.

Exercice 4. Fractale de Newton

Sachant qu'il va falloir appliquer la méthode de Newton un nombre très important de fois, il faut (sous peine de devoir patienter longtemps) choisir un critère de convergence pas trop exigeant ; par exemple la propriété : $|u_n - u_{n-1}| \leqslant 10^{-4}$. En outre, si cette propriété n'est pas vérifiée après 100 itérations, on considèrera que la suite ne converge pas.

Après avoir déterminé que $u_{n+1} = \dfrac{2u_n^3 - a^2 + 1/4}{3u_n^2 - a^2 - 3/4}$ on définit la fonction :

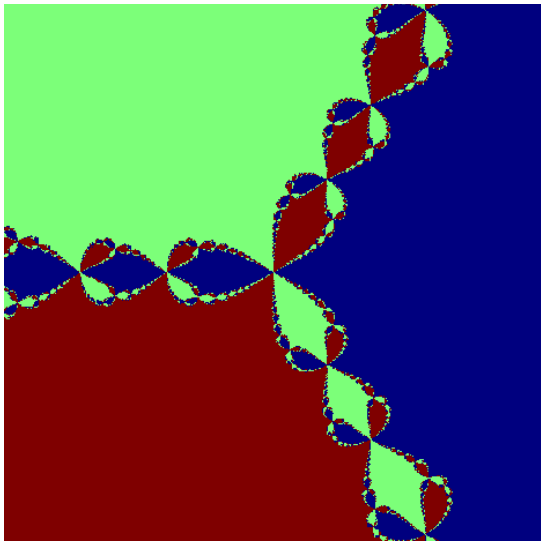
```
def newton(z, a, n=100):
    u = z
    for i in range(n):
        u, v = (2*u**3-a**2+.25)/(3*u**2-a**2-.75), u
        if abs(v-u) < .001:
            return u
    raise ValueError
```

Il reste à définir la fonction permettant de tracer la fractale :

```
def fractale(n, a, xmin=-2, xmax=2):
    r1, r2, r3 = 1, -1/2+a, -1/2-a
    grd = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
            z = complex(xmin+i/n*(xmax-xmin), xmax-j/n*(xmax-xmin))
            try:
                s = newton(z, a)
                if abs(s-r1) < .001:
                    c = 1
                elif abs(s-r2) < .001:
                    c = 2
                elif abs(s-r3) < .001:
                    c = 3
                else:
                    c = 0
            except (ZeroDivisionError, ValueError):
                c = 0
            grd[j,i] = c
    plt.matshow(grd)
    plt.show()
```

On obtient la fractale de Newton lorsque $a = i\dfrac{\sqrt{3}}{3}$, c'est-à-dire lorsque $\mathrm{P} = \mathrm{X}^3 - 1$:

```
>>> fractale(512, complex(0, 1/np.sqrt(3)))
```



Les trois zones colorées sont les bassins d'attraction des trois racines 1, $\mathrm{e}^{2i\pi/3}$ et $\mathrm{e}^{-2i\pi/3}$; ces trois bassins s'étendent autour de chacune des racines mais, plus surprenant, s'entremêlent de manière inextricable entre ces trois grandes zones.

La valeur $a = -0,00508 + 0,33136i$ est plus surprenante encore car outre les trois bassins d'attraction, on voit apparaître des zones dans lesquelles la suite diverge. En particulier, la zone qui se trouve au voisinage de 0 a une forme qui évoque une fractale classique appelée le lapin de Douady :

```
>>> fractale(512, complex(-0.00508, .33136))
>>> fractale(512, complex(-0.00508, .33136), xmin=-.1, xmax=.1)
```