Analyse

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

Exercice 1

```
from scipy.integrate import quad

def u(n):
    def f(x):
        return (3 * x ** 2 - 2 * x ** 3) ** n

    return quad(f, 0, 1)[0]
```

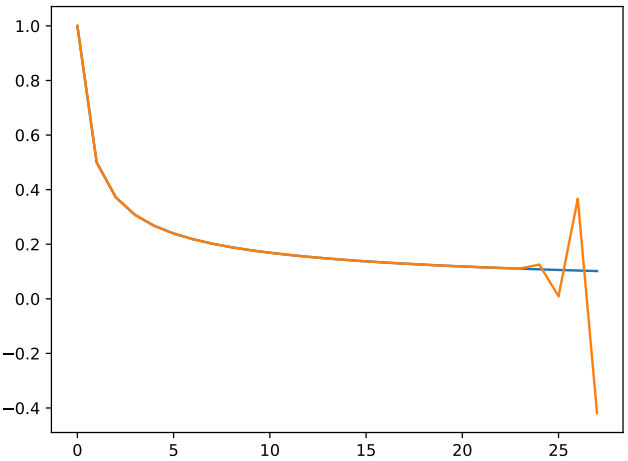
```
from math import comb

def ubis(n):
    s = 0
    for k in range(n + 1):
        s += comb(n, k) * 3 ** k * (-2) ** (n - k) / (3 * n - k + 1)
    return s
```

Le tracé pour $n = 50$ montre que la deuxième version présente des aberrations numériques à partir d'un certain rang (qui démarrent un peu avant $n = 25$ dans ma configuration). Il est donc préférable d'utiliser la première version de la fonction.

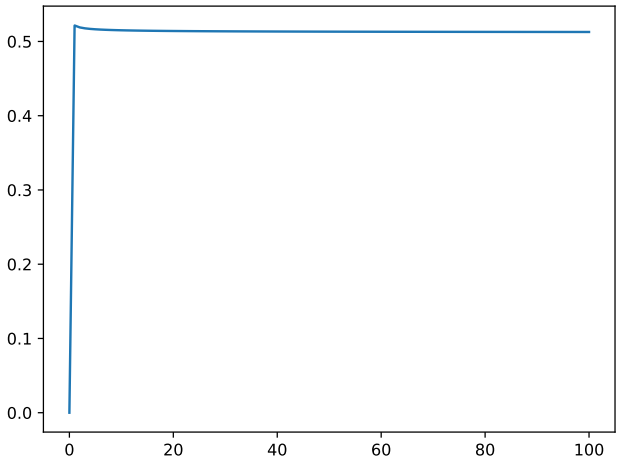
```
U = [u(n) for n in range(28)]
Ubis = [ubis(n) for n in range(28)]

plt.plot(U)
plt.plot(Ubis)
plt.show()
```



```
V = [np.sqrt(100 * k) * u(100 * k) for k in range(101)]

plt.plot(V)
plt.show()
```



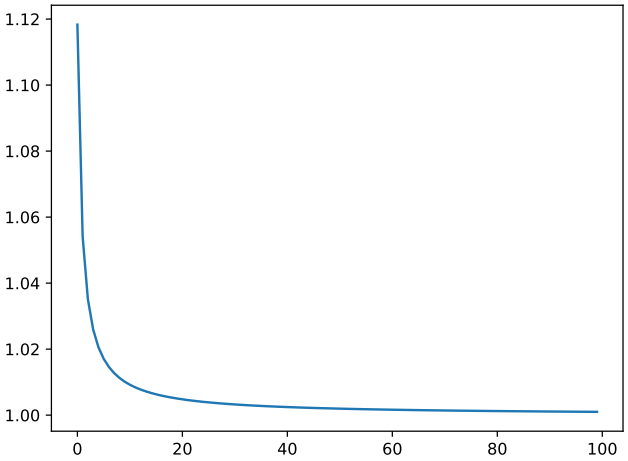
On peut raisonnablement conjecturer qu'il existe une constante K telle que $u_n \sim \dfrac{\mathrm{K}}{\sqrt{n}}$, avec $\mathrm{K} \approx \dfrac{1}{2}$.

Exercice 2 L'expérience montre que la résolution de l'équation : $x = n \ln x$ souffre moins d'aberrations numériques que l'équation $e^x = x^n$.

```
from scipy.optimize import fsolve

def u(n):
    def f(x):
        return x - n * np.log(x)
    return fsolve(f, 1.5)
```

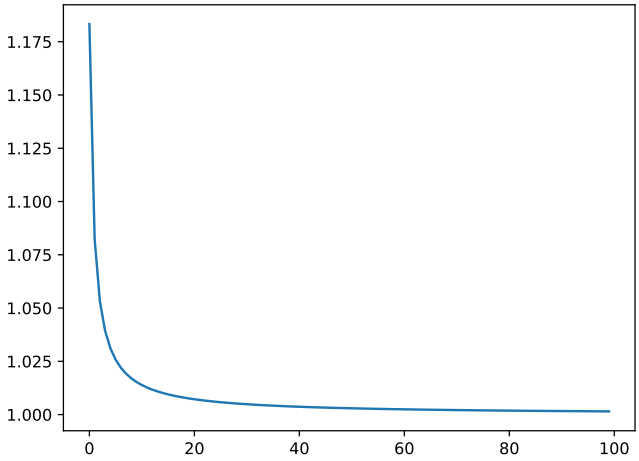
```
U = [u(10 * k) for k in range(1, 101)]
plt.plot(U)
plt.show()
```



Il semble que la suite (u_n) converge vers 1.

On conjecture d'abord la valeur de a en estimant $\lim n(u_n - 1)$:

```
A = [10 * k * (u(10 * k) - 1) for k in range(1, 101)]
plt.plot(A)
plt.show()
```



Il semble bien que $a = 1$. On procède de même pour b, qui donne cette fois-ci $b = 3/2$ (avec un graphe analogue, non reporté ici) :

```
B = [(10 * k) ** 2 * (u(10 * k) - 1 - 1 / (10 * k)) for k in range(1, 101)]
plt.plot(B)
plt.show()
```

Exercice 3

```
from scipy.integrate import odeint

a = np.sqrt(np.pi / 2)
```

Le tracé de f ne pose pas de problème :

```
def f(t):
    return np.sin(t * t) - np.cos(t * t) / (2 * a)

T = np.linspace(a, 4, 128)
G1 = [f(t) for t in T]
plt.plot(T, G1)
```

Pour utiliser `odeint` il faut commencer par définir la fonction à deux variables (la première étant vectorielle) $\mathrm{F}(\mathrm{X}, t)$ où $\mathrm{X} = (x, x')$ et $\mathrm{F}(\mathrm{X}, t) = (x'', x')$:

```
def F(X, t):
    x, xprime = X
    return xprime, (xprime - 4 * t ** 3 * x) / t
```

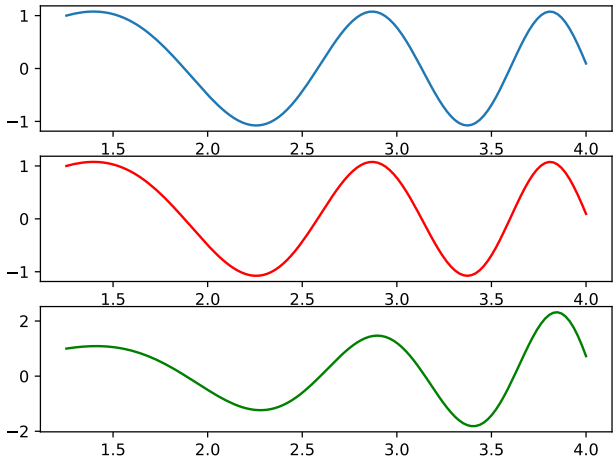
La fonction `odeint` prend pour arguments F, la condition initiale $(x(a), x'(a))$ et une discrétisation du temps. Elle renvoie un vecteur dont les composantes sont les valeurs approchées $(x(t_i), x'(t_i))$; il ne faut garder que les premières composantes pour tracer la fonction x.

```
M = odeint(F, [1, 1], T)
G2 = [m[0] for m in M]
plt.plot(T, G2)
```

Enfin, la méthode d'Euler consiste à faire terme par terme ce que produit la fonction `odeint` en utilisant les approximations $x(t_{i+1}) \approx x(t_i) + (t_{i+1} - t_i)x'(t_i)$ et $x'(t_{i+1}) \approx x'(t_i) + (t_{i+1} - t_i)x''(t_i)$.

```
M = [[1, 1]]
for i in range(0, len(T) - 1):
    x, xprime = M[-1]
    xseconde = (xprime - 4 * T[i] ** 3 * x) / T[i]
    y = x + xprime * (T[i + 1] - T[i])
    yprime = xprime + xseconde * (T[i + 1] - T[i])
    M.append([y, yprime])
G3 = [m[0] for m in M]
plt.plot(T, G3)
```

On obtient les tracés suivants :



Exercice 4

```
from numpy.polynomial import Polynomial

def Q(p):
    lst = [-k - 1 for k in range(2 * p)] + [0, 2 * p + 2]
    return Polynomial(lst)
```

Le code qui suit permet de conjecturer que Q_p possède une seule racine réelle :

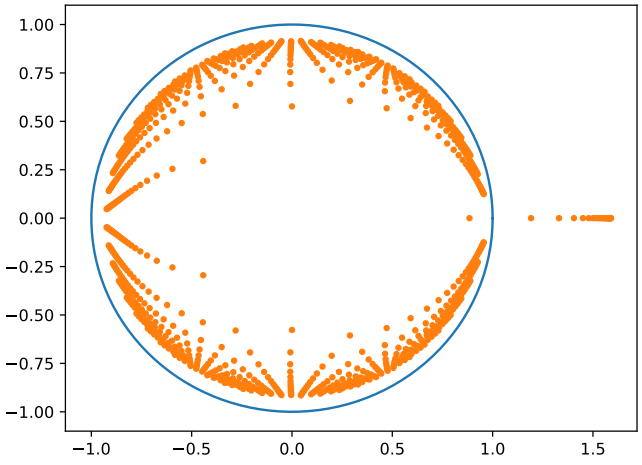
```
for p in (3, 4, 5, 10):
    print(Q(p).roots())
```

Le tracé suivant permet de conjecturer que toutes les racines complexes non réelles de Q_p sont incluses dans le disque unité :

```
X = [np.cos(t) for t in np.linspace(0, 2 * np.pi, 128)]
Y = [np.sin(t) for t in np.linspace(0, 2 * np.pi, 128)]
plt.plot(X, Y)

Rx, Ry = [], []
for p in range(1, 30):
    for z in Q(p).roots():
        Rx.append(z.real)
        Ry.append(z.imag)
plt.plot(Rx, Ry, ".")

plt.show()
```


1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.0
−0.5
0.0
0.5
1.0
1.5