Corrigé

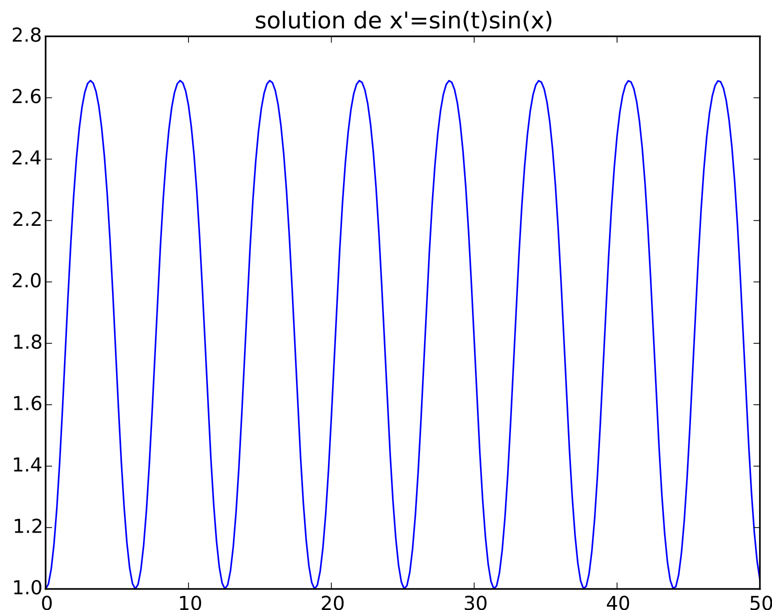
Résolution numérique d'une équation différentielle

Exercice 1. On commence par importer les différents modules et fonctions dont nous auront besoin :

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint
```

La fonction `odeint` nous permet d'obtenir une résolution numérique de référence pour l'équation différentielle qui nous intéresse :

```
def f(x, t):
    return np.sin(t) * np.sin(x)

t = np.linspace(0, 50, 256)
x = odeint(f, 1, t)
plt.plot(t, x)
plt.title("solution de x'=sin(t)sin(x)")
plt.show()
```

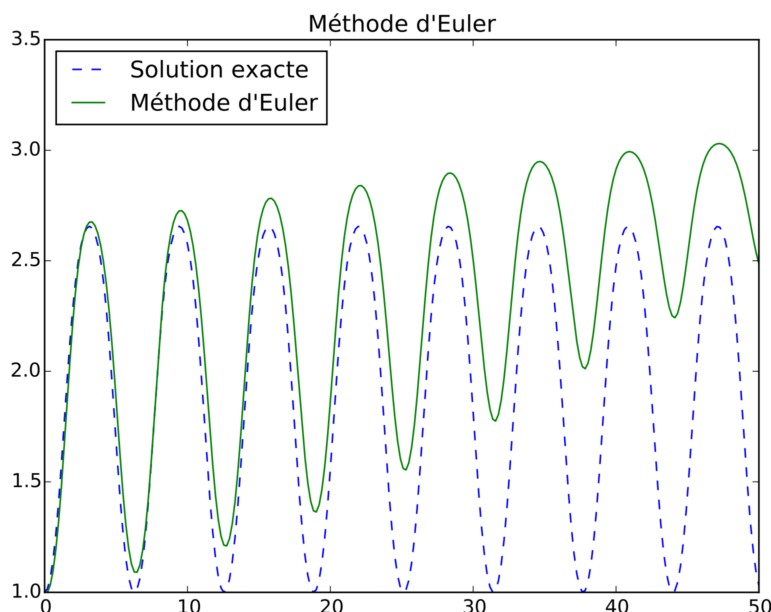


La méthode d'Euler se définit ainsi :

```
def euler(f, x0, t):
    n = len(t)
    x = [x0]
    for k in range(0, n-1):
        h = t[k+1] - t[k]
        p1 = f(x[k], t[k])
        x.append(x[k] + h * p1)
    return x
```

mais le résultat est assez décevant, la solution fournie par la méthode s'éloigne irrémédiablement de la vraie solution :

```python
x1 = euler(f, 1, t)
plt.plot(t, x, '--', label='Solution exacte')
plt.plot(t, x1, label="Méthode d'Euler")
plt.title("Méthode d'Euler")
plt.legend(loc='upper left')
plt.show()
```



En revanche, les méthodes de Heun et RK_4 s'avèrent bien plus précises :

```python
def heun(f, x0, t):
    n = len(t)
    x = [x0]
    for k in range(0, n-1):
        h = t[k+1] - t[k]
        p1 = f(x[k], t[k])
        p2 = f(x[k] + h * p1, t[k+1])
        x.append(x[k] + h * (p1 + p2) / 2)
    return x
```

```python
def rk4(f, x0, t):
    n = len(t)
    x = [x0]
    for k in range(0, n-1):
        h = t[k+1] - t[k]
        p1 = f(x[k], t[k])
        p2 = f(x[k] + h * p1 / 2, t[k] + h / 2)
        p3 = f(x[k] + h * p2 / 2, t[k] + h / 2)
        p4 = f(x[k] + h * p3, t[k+1])
        x.append(x[k] + h * (p1+2*p2+2*p3+p4) / 6)
    return x
```

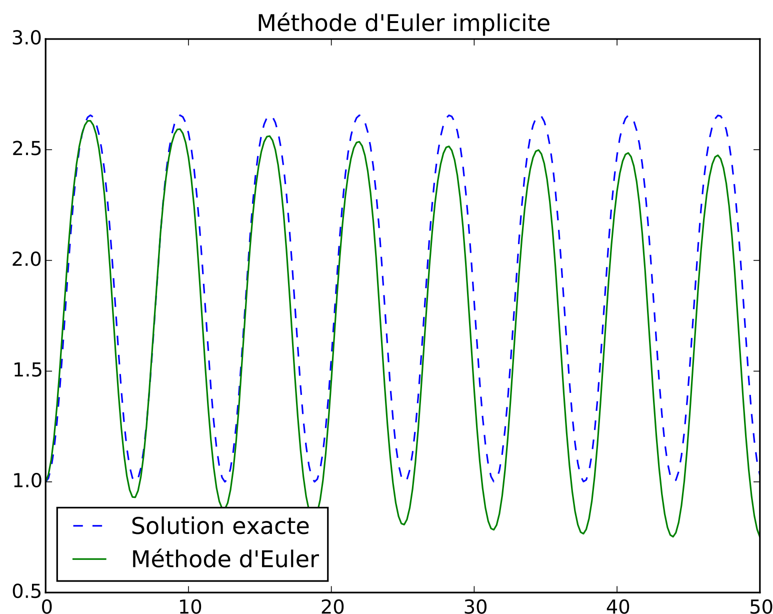
et fournissent des résultats peu discernables de la solution exacte.

Enfin, la méthode d'Euler implicite est définie par :

```python
from scipy.optimize import newton

def eulerbis(f, x0, t):
    n = len(t)
    x = [x0]
    for k in range(0, n-1):
        h = t[k+1] - t[k]
        s = newton(lambda u: u - x[k] - f(u, t[k+1]) * h, x[k])
        x.append(s)
    return x
```

Elle fournit un résultat visuellement un peu meilleur que la méthode d'Euler classique, mais s'éloigne elle aussi de la solution exacte :

```
    x4 = eulerbis(f, 1, t)
    plt.plot(t, x, '--', label='Solution exacte')
    plt.plot(t, x4, label="Méthode d'Euler")
    plt.title("Méthode d'Euler implicite")
    plt.legend(loc='best')
    plt.show()
```



Exercice 2. On définit l'erreur de la méthode ainsi :

```python
def erreur(methode, n):
    t = np.linspace(0, 2, n)
    def f(x, t): return x
    x = methode(f, 1, t)
    m = 0
    for k in range(n):
        m = max(m, abs(x[k] - np.exp(t[k])))
    return m
```

La recherche du rang minimal pour une précision donnée peut être réalisée par une méthode dichotomique, à condition de posséder une valeur n_0 qui réalise cette précision (valeur qu'on peut obtenir en tâtonnant). On définit donc la fonction :

```python
def rang(methode, epsilon, n0):
    if erreur(methode, n0) > epsilon:
        return None
    a, b = 2, n0
    while b - a > 1:
        c = (a + b) // 2
        if erreur(methode, c) > epsilon:
            a = c
        else:
            b = c
    return b
```

Cette fonction fournit les résultats suivants :

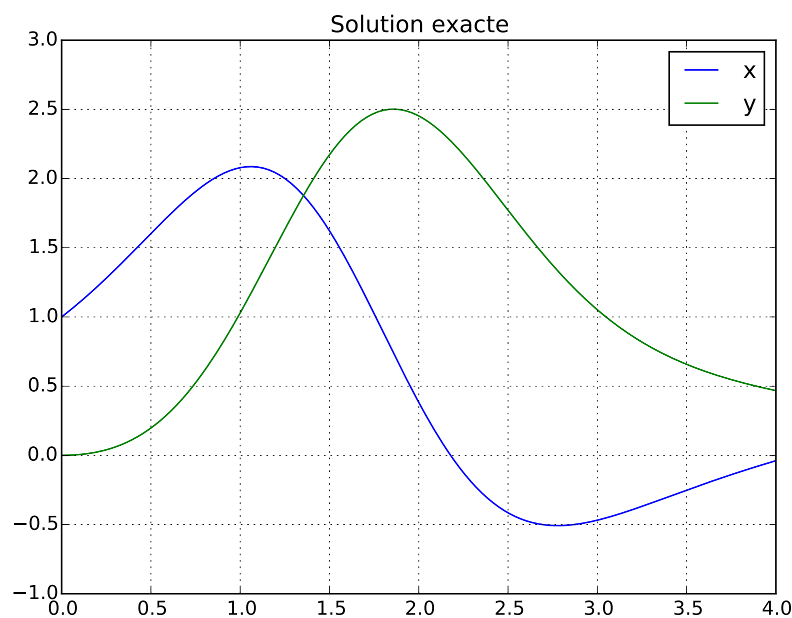
```
>>> rang(euler, 1e-1, 200)
147
>>> rang(euler, 1e-2, 2000)
1477
>>> rang(euler, 1e-3, 20000)
14777
```

```
>>> rang(heun, 1e-2, 100)
32
>>> rang(heun, 1e-4, 1000)
315
>>> rang(heun, 1e-6, 10000)
3140
```

```
>>> rang(rk4, 1e-4, 100)
13
>>> rang(rk4, 1e-8, 200)
120
>>> rang(rk4, 1e-12, 1500)
1187
```

Exercice 3. On obtient la solution numérique de ce système à l'aide du script :

```
def F(X, t):
    [x, y] = X
    return [np.cos(t) * x - np.sin(t) * y, np.sin(t) * x + np.cos(t) * y]

t = np.linspace(0, 4, 256)
X = odeint(F, [1, 0], t)
x, y = X[:, 0], X[:, 1]
plt.plot(t, x, label='x')
plt.plot(t, y, label='y')
plt.legend(loc='best')
plt.title('Solution exacte')
plt.grid()
plt.show()
```

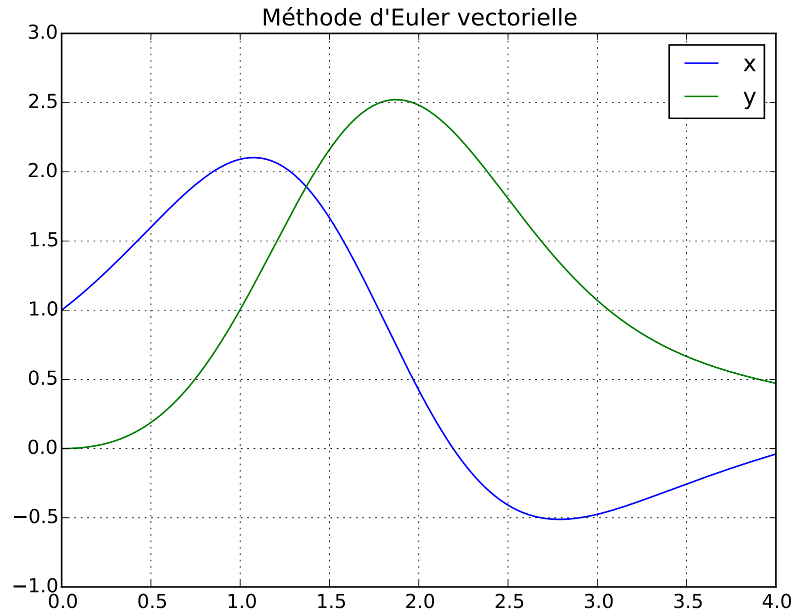


La méthode d'Euler vectorielle peut se définir ainsi :

```
def euler_vect(F, X0, t):
    n = len(t)
    X = [X0]
    for k in range(0, n-1):
        h = t[k+1] - t[k]
        p1 = F(X[k], t[k])
        X.append([X[k][0] + h * p1[0], X[k][1] + h * p1[1]])
    return X
```

et le graphe obtenu est très proche du graphe exact :

```
X1 = euler_vect(F, [1, 0], t)
x1 = [z[0] for z in X1]
y1 = [z[1] for z in X1]
plt.plot(t, x1, label='x')
plt.plot(t, y1, label='y')
plt.legend(loc='best')
plt.title("Méthode d'Euler vectorielle")
plt.grid()
plt.show()
```



On peut noter une légère différence entre les deux scripts pour définir x et y ; cette différence est due au fait que dans le premier script, X est un tableau NUMPY qui permet une indexation plus aisée des tableaux bi-dimensionnels.

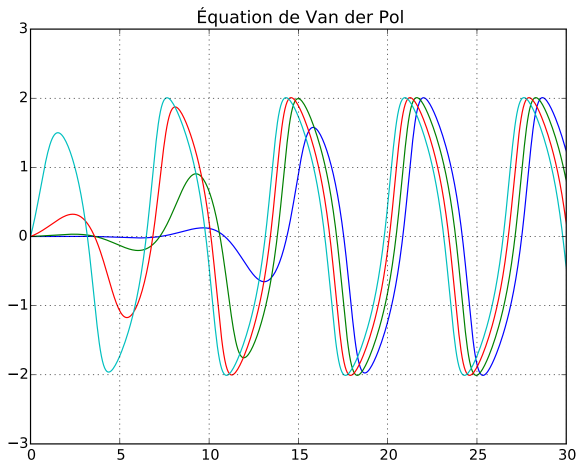
Exercice 4. Équation de Van der Pol

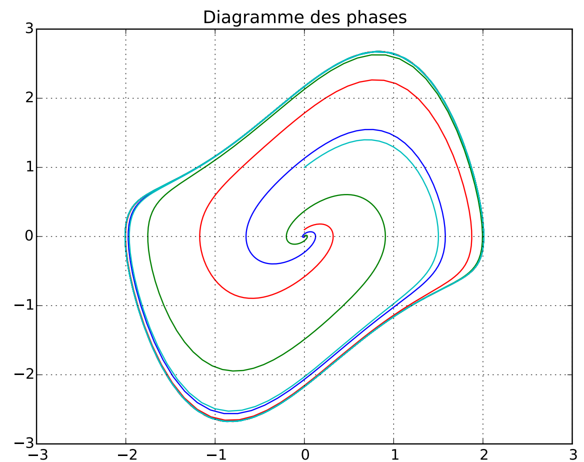
Commençons par définir la fonction qui caractérise l'équation différentielle :

```
def f(X, t):
    x, dx = X
    return [dx, mu * (1 - x * x) * dx - x]
```

On obtient ensuite les deux graphes demandés à l'aide du script :

```
t = np.linspace(0, 30, 512)

mu = 1
for v in [.001, .01, .1, 1]:
    X = odeint(f, [0, v], t)
    plt.figure(1)
    plt.plot(t, X[:, 0])
    plt.figure(2)
    plt.plot(X[:, 0], X[:, 1])

plt.figure(1)
plt.title('Équation de Van der Pol')
plt.grid()

plt.figure(2)
plt.title('Diagramme des phases')
plt.grid()

plt.show()
```

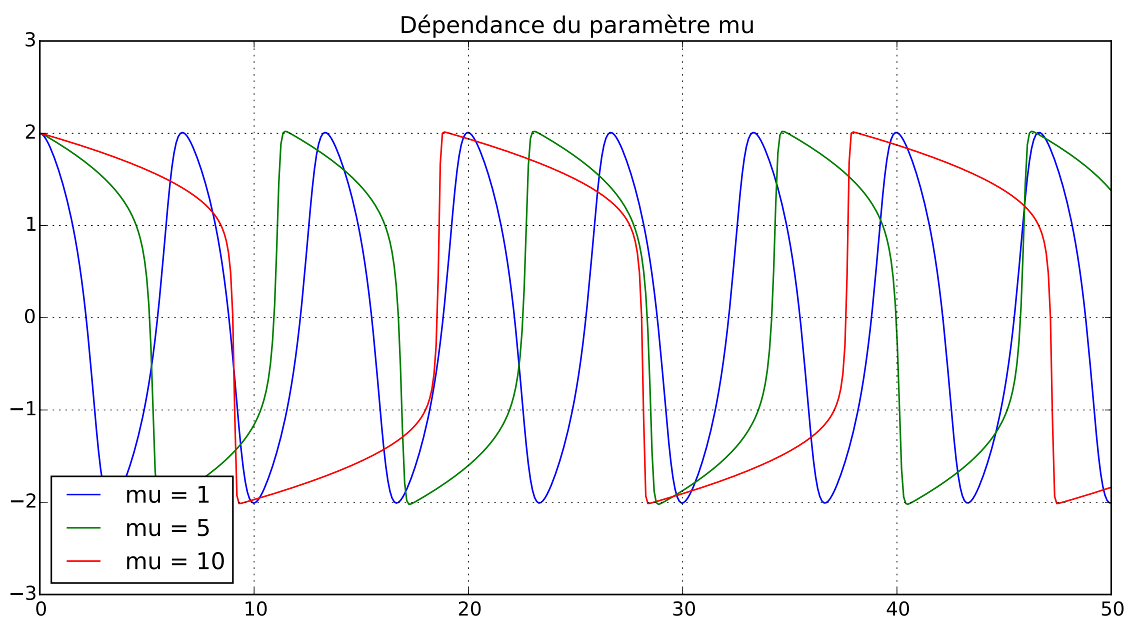




On observent que les solutions convergent vers un régime périodique indépendant (à un déphasage près) des conditions initiales.

Nous allons maintenant constater qu'en jouant sur le paramètre μ, il est possible d'obtenir des solutions sensiblement non sinusoïdales :

```
plt.figure(3, figsize=(12,6))
t = np.linspace(0, 50, 512)
for mu in [1, 5, 10]:
    X = odeint(f, [2, 0], t)
    plt.plot(t, X[:, 0], label='mu = {}'.format(mu))
plt.title('Dépendance du paramètre mu')
plt.legend(loc='lower left')
plt.show()
```

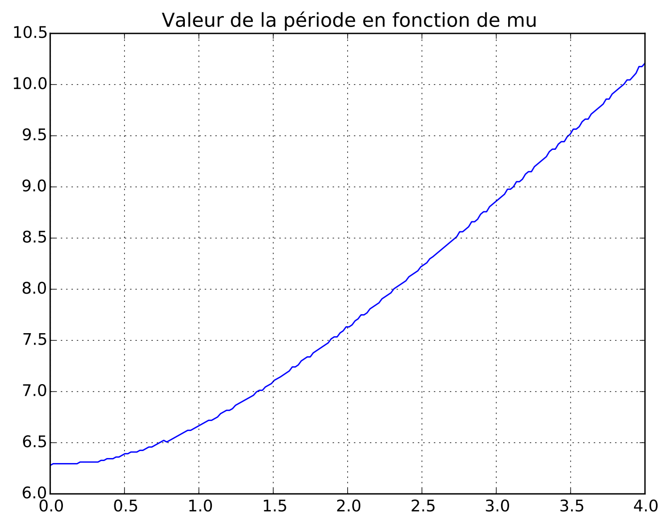


On peut observer que le phénomène de relaxation est d'autant plus marqué que μ augmente.
Par ailleurs, il apparaît que la période dépend du paramètre μ. Pour calculer celle-ci, on calcule la moyenne des écarts entre deux maximums consécutifs à l'aide de la fonction :

```python
def periode(t, x):
    s = []
    for k in range(1, len(t)-1):
        if x[k-1] < x[k] and x[k+1] < x[k]:
            s.append(t[k])
    p = 0
    for k in range(1, len(s)):
        p += s[k]-s[k-1]
    return p / (len(s)-1)
```

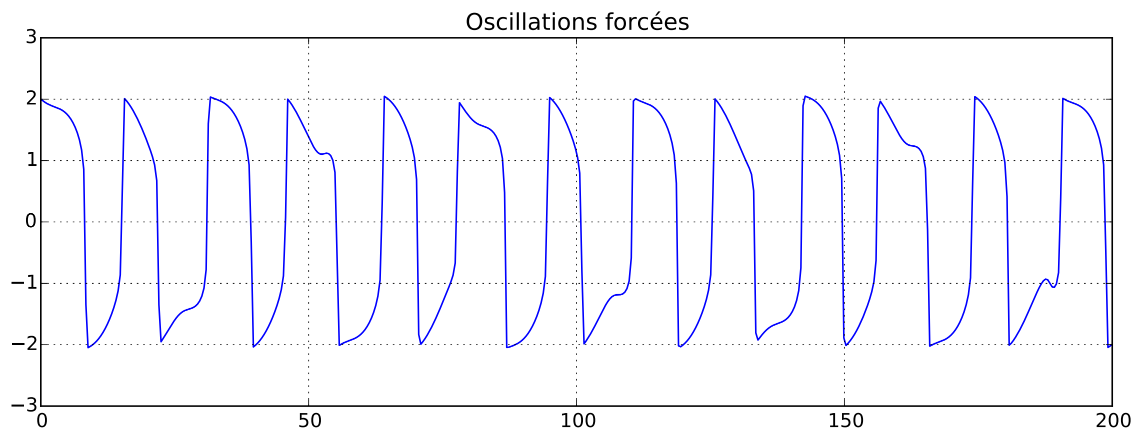
En faisant varier μ entre 0 et 4 on obtient le graphe des périodes suivant :

```python
plt.figure(4)
mus = np.linspace(0, 4, 200)
per = []
for mu in mus:
    x = odeint(f, [2, 0], t)
    per.append(periode(t, x[:, 0]))
plt.plot(mus, per)
plt.title('Valeur de la période en fonction de mu')
plt.grid()
plt.show()
```



Enfin, l'excitation de cet oscillateur par un terme harmonique permet d'observer que l'amplitude de l'onde est indépendante de la force extérieure appliquée, avec néanmoins un comportement chaotique.

```python
mu = 8.53
A = 1.2
omega = .1

def g(X, t):
    x, dx = X
    return [dx, mu * (1 - x * x) * dx - x + A * np.sin(2 * np.pi * omega * t)]

plt.figure(5, figsize=(12,4))
t = np.linspace(0, 200, 500)
X = odeint(g, [2, 0], t)
plt.plot(t, X[:, 0])
plt.title('Oscillations forcées')
plt.grid()
plt.show()
```


Oscillations forcées
3
2
1
0
−1
−2
−3
0
50
100
150
200