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

2017/2018

TP n° 10 : Équations différentielles

Correction de l'exercice 1 – On commence par importer les modules nécessaires et définir nos deux fonctions tests :

```
import math
import matplotlib.pyplot as plt
import scipy.integrate as si
import numpy as np

def F1(y,t):
    return y

def F2(y,t):
    return math.sin(t) * math.sin(y)
```

Voici la méthode d'Euler. On progresse de pas en pas en approchant à chaque étape la courbe par la droite issue de (x, y) de pente $f(y, x)$ (valeur approchée de y').

```
def euler(f,a,b,y0,n):
    """ résolution de y'=f(y,t) par la méthode d'Euler """
    h = (b-a) / n
    X = [a]
    Y = [y0]
    x = a
    y = y0
    for i in range(1,n+1):
        y += h * f(y,x)
        x += h
        X.append(x)
        Y.append(y)
    return (X,Y)
```

La méthode RK2 est obtenue en remplaçant cette pente par la pente approchée au milieu de l'intervalle, extrapolée à la façon d'Euler :

```
def RK2(f,a,b,y0,n):
    """ résolution de y'=f(y,t) par la méthode RK2 """
    h = (b-a) / n
    X = [a]
    Y = [y0]
    x = a
    y = y0
    for i in range (1,n+1):
        z = y + h / 2 * f(y,x)
        p = f(z,x+h / 2)
        y += h * p
        x += h
        X.append(x)
```

```
        Y.append(y)
    return (X,Y)
```

Enfin, la méthode RK4 (la méthode de Runge-Kutta classique) consiste à faire une moyenne des pentes successives obtenue en x, $x+\frac{h}{2}$, $x+\frac{h}{2}$ et $x+h$, ces pentes étant successivement obtenues en approchant la courbe depuis x en utilisant la pente précédente (ou lapente initiale pour la première). La moyenne est effectuée avec les pondrations $\frac{1}{6}, \frac{2}{6}, \frac{2}{6}, \frac{1}{6}$.

```
def RK4(f,a,b,y0,n):
    """ résolution de y'=f(y,t) par la méthode RK4 """
    h = (b-a) / n
    X = [a]
    Y = [y0]
    x = a
    y = y0
    for i in range (1,n+1):
        p1 = f(y,x)
        y2 = y + h / 2 * p1
        p2 = f(y2,x + h/2)
        y3 = y + h / 2 * p2
        p3 = f(y3,x + h/2)
        y4 = y + h * p3
        p4 = f(y4,x +h)
        p = (p1 + 2* p2 + 2* p3 + p4)/6
        y += h * p
        x += h
        X.append(x)
        Y.append(y)
    return(X,Y)
```

La fonction ci-dessous effectue le tracé des courbes obtenues par les différentes méthodes, ainsi que par la fonction `odeint`. On trace également la courbe disponible, si elle est fournie (paramètre optionnel)

```
def trace(f,a,b,y0,n, g = False):
    X = [a + i * (b-a) / n for i in range(n+1)]
    if g:
        Y = [g(x) for x in X]
        plt.plot(X,Y, label = 'théorique')
    Y = si.odeint(f, y0, X)
    Y = [ c[0] for c in Y]
    plt.plot(X,Y,label = 'odeint')
    X, Y = euler(f,a,b,y0,n)
    plt.plot(X,Y, label = 'euler')
    X, Y = RK2(f,a,b,y0,n)
    plt.plot(X,Y, label = 'RK2')
    X, Y = RK4(f,a,b,y0,n)
    plt.plot(X,Y, label = 'RK4')
    plt.legend(loc = 'best')
    plt.title("Résolution␣par␣différentes␣méthodes␣de␣.....")
    plt.show()

trace(F1,0,2,1,10, lambda x: math.exp(x))
trace(F1,0,10,1,20, lambda x: math.exp(x))
```

```
trace(F2,0,50,1,100)
trace(F2,0,50,1,200)
```

Les 4 essais donnés fournissent les courbes des figures 1, 2, 3 et 4

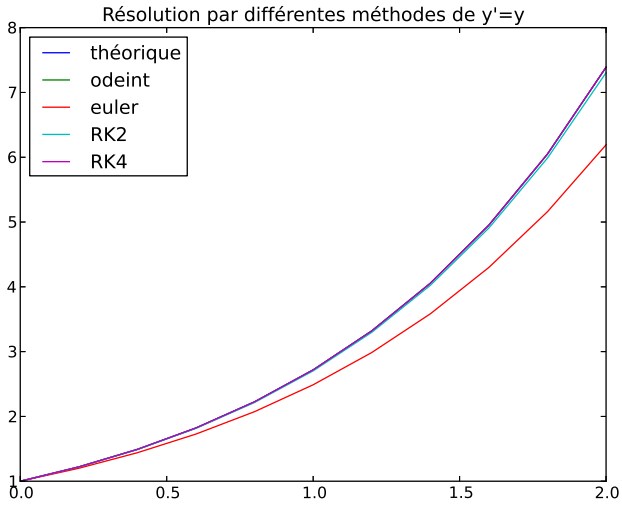

FIGURE 1 – Résolution de $y' = y$ sur $[0, 2]$ avec 10 pas

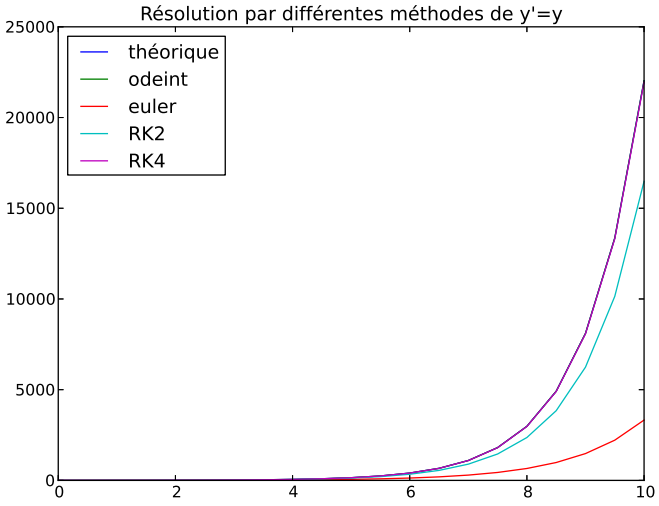

FIGURE 2 – Résolution de $y' = y$ sur $[0, 10]$ avec 20 pas

Les courbes fournies par `odeint`, par la méthode RK4 et la courbe théorique sont pratiquement confondues, même pour des petites valeurs de n. La méthode d'Euler n'est pas très efficace, comme on peut le constater. La méthode RK2 est raisonnable (à l'oeil nu) pour des valeurs raisonnablement grandes de n, mais pas aussi efficace que RK4.
Pour une meilleure étude comparative, on calcule explicitement l'erreur maximale sur l'intervalle $[0, 2]$, dans le cas de la fonction exponentielle.

```
def erreur(methode,f,a,b,y0,n,g):
    """ erreur faite par rapport à la solution théorique g """
    X,Y = methode(f,a,b,y0,n)
    err = 0
```

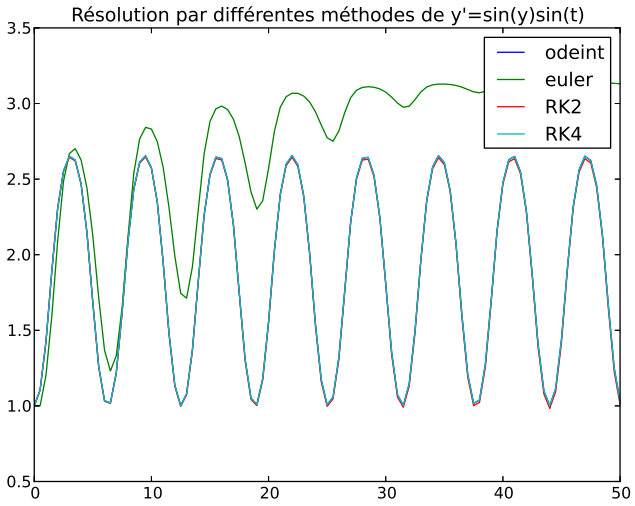

FIGURE 3 – Résolution de $y' = \sin(y)\sin(t)$ sur $[0, 50]$ avec 100 pas

```
    for i in range(len(X)):
        e = abs(Y[i] - g(X[i]))
        if e > err:
            err = e
    return err

def affichage_erreurs(f,a,b,y0,n,g):
    print("Erreurs␣obtenues␣pour␣n␣=␣{}␣:␣".format(n))
    print("Euler␣:␣{}".format(erreur(euler,f,a,b,y0,n,g)))
    print("RK2␣:␣{}".format(erreur(RK2,f,a,b,y0,n,g)))
    print("RK4␣:␣{}".format(erreur(RK4,f,a,b,y0,n,g)))
    print()

affichage_erreurs(F1,0,2,1,100,lambda x: math.exp(x))
affichage_erreurs(F1,0,2,1,200,lambda x: math.exp(x))
```

On obtient les résultats suivants :

```
Erreurs obtenues pour n = 100 :
Euler : 0.144409980678323
RK2 : 0.0009704838383779446
RK4 : 1.9378555649041118e-08

Erreurs obtenues pour n = 200 :
Euler : 0.07303824710072693
RK2 : 0.0002444579508580347
RK4 : 1.2212924005439163e-09
```

On se rend compte que multiplier par 2 semble diviser l'erreur par 2 pour la méthode d'Euler, par $4 = 2^2$ pour la méthode RK2, et par $16 = 2^4$ pour la méthode RK4. Plus précisément, on peut obtenir une évaluation numérique de l'ordre de ces méthodes en calculant le logarithme en base 2 des quotients des valeurs obtenues :

```
def approximation_des_ordres():
    print('Approximation␣des␣ordres␣des␣méthodes:␣')
    for methstr in ['euler', 'RK2','RK4']:
```

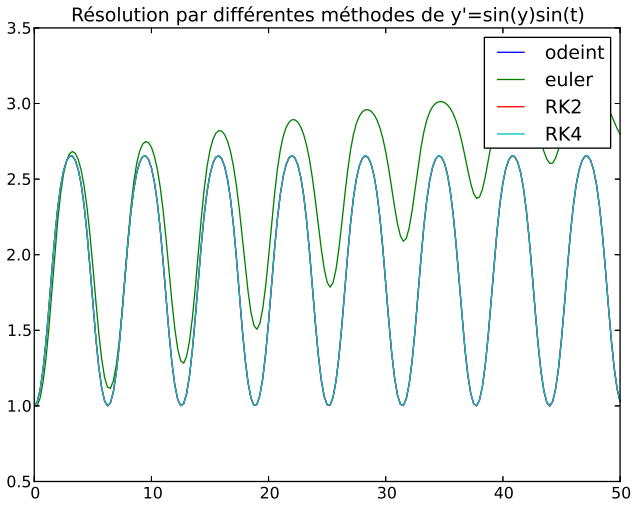

FIGURE 4 – Résolution de $y' = \sin(y)\sin(t)$ sur $[0, 50]$ avec 200 pas

```
        meth = eval(methstr)
        ordre= erreur(meth,F1,0,2,1,100,math.exp) /erreur(meth,F1,0,2,1,200,math.exp)
        print("ordre␣" + methstr +"␣:␣{}".format (math.log(ordre,2)))
    print()

approximation_des_ordres()
```

On obtient, comme attendu :

```
Approximation des ordres des méthodes:
ordre euler : 0.9834464088280334
ordre RK2 : 1.9891178587506042
ordre RK4 : 3.987980490370499
```

Pour terminer, on calcule, pour chaque méthode, le rang nécessaire pour obtenir une approximation uniforme de l'exponentielle à ε près sur l'intervalle $[0, 2]$:

```
def rang_necessaire(methode,f,a,b,y0,g,eps):
    """ rang nécessaire n pour obtenir une approximation uniforme à eps
    près """
    n = 1
    while erreur(methode,f,a,b,y0,n,g) > eps:
        n += 1
    return n

print('euler,␣1e-2:␣{}'.format(rang_necessaire(euler,F1,0,2,1,math.exp,1e-2)))
print('RK2,␣1e-2:␣{}'.format(rang_necessaire(RK2,F1,0,2,1,math.exp,1e-2)))
print('RK4,␣1e-2:␣{}'.format(rang_necessaire(RK4,F1,0,2,1,math.exp,1e-2)))
print('RK2,␣1e-6:␣{}'.format(rang_necessaire(RK2,F1,0,2,1,math.exp,1e-6)))
print('RK4,␣1e-6:␣{}'.format(rang_necessaire(RK4,F1,0,2,1,math.exp,1e-6)))
print('RK4,␣1e-12:␣{}'.format(rang_necessaire(RK4,F1,0,2,1,math.exp,1e-12)))
```

Les résultats :

```
Rang nécessaire pour une approximation uniforme:
```

```
euler, 1e-2: 1476
RK2, 1e-2: 31
RK4, 1e-2: 4
RK2, 1e-6: 3139
RK4, 1e-6: 38
RK4, 1e-12: 1097
```

Les résultats sont conformes à ceux attendus : la méthode d'Euler ne fournit par de très bonnes approximations. La méthode RK2 sature autour de 10^{-8}, et RK4 peut nous fournir en temps raisonnable des approximations à 10^{-12} ou même encore mieux.

Correction de l'exercice 2 – On commence par importer les modules, et à définir la fonction vectorielle définissant l'équation différentielle, en remarquant que celle-ci s'écrit :

$$\begin{pmatrix} x'(t) \\ x''(t) \end{pmatrix} = \begin{pmatrix} x'(t) \\ -\sin(x(t) - \frac{1}{4}x'(t) \end{pmatrix},$$

c'est-à-dire $Y' = F(Y,t)$, où

$$Y = \begin{pmatrix} x \\ x' \end{pmatrix} \qquad \text{et} \qquad F\left(\begin{pmatrix} y_0 \\ y_1 \end{pmatrix}, t\right) = \begin{pmatrix} y_1 \\ -\sin(y_0) - \frac{y_1}{4} \end{pmatrix}.$$

On fait ressortir les données vectorielles sous forme d'une liste, afin d'être conforme avec la syntaxe de `odeint`.

```
import math
import scipy.integrate as si
import numpy as np
import matplotlib.pyplot as plt

def F(Y,t):
    return [Y[1], -math.sin(Y[0]) - Y[1] / 4]
```

La résolution de l'équation du pendule se fait alors avec `odeinte`. On effectue le tracé pour les valeurs de v entières de 1 à 10.

```
def pendule(v,tmax):
    T = np.linspace(0,30,301)
    Z = si.odeint(F,[0,v],T)
    X = [z[0] for z in Z]
    V = [z[1] for z in Z]
    return X, V, T

def trace_angle(vmax,tmax):
    for v in range(1, vmax+1):
        X,V,T= pendule(v,tmax)
        plt.plot(T,X,label='v={}'.format(v))
    plt.legend()
    plt.xlabel('temps')
    plt.ylabel('position')
    plt.grid()
    plt.title('Position␣en␣fonction␣du␣temps')
    plt.savefig('py043-1.eps')
    plt.show()
```

On obtient le graphe de la figure 5

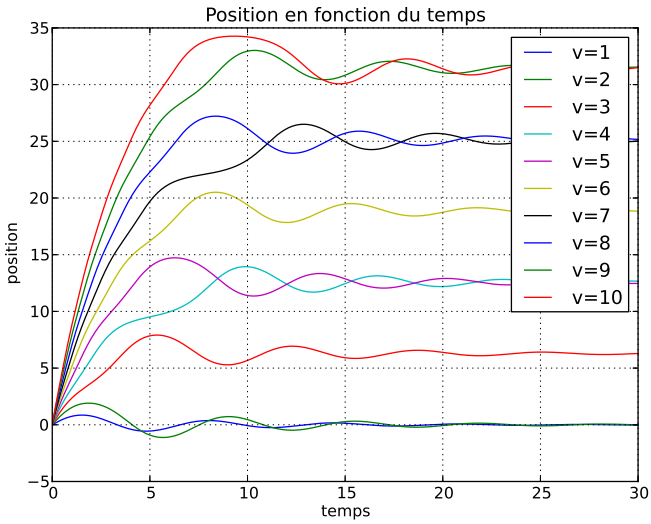

FIGURE 5 – Position du pendule en fonction du temps

On se rend compte que les courbes de x semblent se stabiliser autour de valeurs $2k\pi$. La valeur de k est le nombre de tours complets effectués par le pendule avant de se stabiliser.
On trace également le diagramme de phase pour plusieurs valeurs initiales de v, fournies dans un tableau :

```
def diagramme_phase(V,tmax):
    """ trace le diagramme de phase pour les valeurs initiales de v
    dans le tableau V"""
    for v in V:
        X,V,T= pendule(v,tmax)
        plt.plot(X,V,label ='v={}'.format(v))
    plt.xlabel('position')
    plt.ylabel('vitesse')
    plt.grid()
    plt.title('Diagramme de phase')
    plt.legend()
    plt.savefig('py043-2.eps')
    plt.show()

diagramme_phase([2,3,4,6],30)
```

On obtient les graphes de la figure refphase. Dans ce diagramme, on voit bien le ralentissement de la vitesse lorsque le pendule remonte (angles compris entre $2k\pi$ et $(2k+1)\pi$ parcourus dans l'ordre croissante), l'augmentation de la vitesse sur les autres intervalles, ceci tant qu'on fait des tours complets, puis une oscillation de la vitesse lors de la stabilisation.
On recherche la valeur minimale de la vitesse initiale v à donner au pendule pour que le pendule fasse k tours avant de se stabiliser. On remarque que pour une vitesse initiale donnée, le nombre de tours k est l'unique entier tel que pour t assez grand, $x(t) \in [(2k-1)\pi, (2k+1)\pi]$. On fait une résolution sur un intervalle $[0, t_m]$ qu'on suppose assez grand pour que le pendule soit stabilisé dans le bon intervalle. Pour savoir si on a effectué k tours complets, il suffit alors de comparer la dernière valeur calculée de x (donc $x(t_m)$) à la valeur $(2k-1)\pi$.
On effectue alors un premier balayage grossier (en faisant croître les vitesses de 1 en 1), afin de trouver un encadrement de la vitesse à donner. On affine la précision à l'aide d'une dichotomie.

```
def vitessemin(k,err):
    v = 1
```

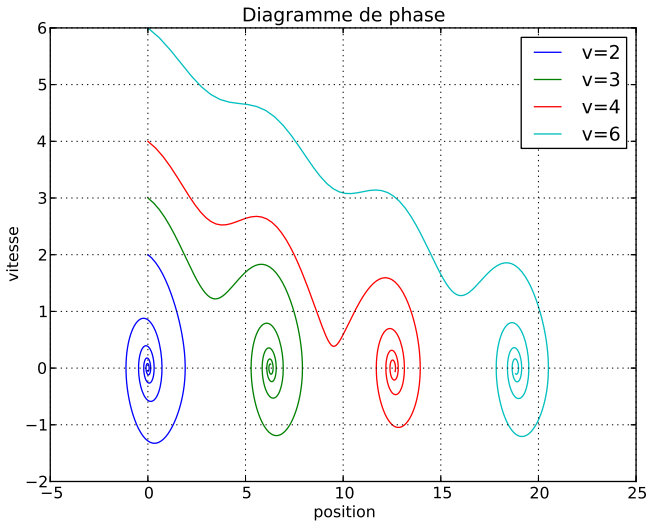

FIGURE 6 – Diagramme de phase

```
    X,V,T= pendule(v,100)
    while X[-1] < (2 * k -1) *math.pi:
        v += 1
        X,V,T= pendule(v,100)
    vmax = v
    vmin = v-1
    while vmax - vmin > err:
        v = (vmax + vmin) / 2
        X,V,T= pendule(v,100)
        if X[-1] < (2 * k -1) *math.pi:
            vmin = v
        else:
            vmax = v
    return vmax

for k in range(10):
    print('Pour faire {} tours: v0 = {}'.format(k,vitessemin(k, 1e-10)))
```

Les résultats obtenus :

```
Pour faire 0 tours: v0 = 5.820766091346741e-11
Pour faire 1 tours: v0 = 2.523139135155361
Pour faire 2 tours: v0 = 3.9259209479787387
Pour faire 3 tours: v0 = 5.421297665743623
Pour faire 4 tours: v0 = 6.949933790077921
Pour faire 5 tours: v0 = 8.493937145278323
Pour faire 6 tours: v0 = 10.046239296207204
Pour faire 7 tours: v0 = 11.603515822382178
Pour faire 8 tours: v0 = 13.164003947807942
Pour faire 9 tours: v0 = 14.726679210725706
```

Correction de l'exercice 3 – On importe les modules, et on définit la fonction donnant l'équation différentielle. Afin de gérer les paramètres (qui ne doivent pas être passés en variable dans `odeint`), on crée une fonction des paramètres,

retournant la fonction correspondante des variables Y et t.

```
import math
import scipy.integrate as si
import matplotlib.pyplot as plt
import numpy as np

def F(mu):
    return lambda Y,t: [Y[1],mu * (1- Y[0]**2)*Y[1] - Y[0]]
```

La résolution de l'équation de Van der Pol :

```
def vanderpol(mu,x0,xp0):
    T = np.linspace(0,30,301)
    Z = si.odeint(F(mu),[x0,xp0],T)
    X = [z[0] for z in Z]
    Y = [z[1] for z in Z]
    return X,Y,T
```

Le tracé de plusieurs courbes de x en fonction du temps, pour des paramètres passés sous forme de liste (on fait le tracé pour toutes les valeurs de la liste, pour chacun des paramètres), ainsi que les diagrammes de phase :

```
def tracex(Mu,X0,Xp0):
    for mu in Mu:
        for x0 in X0:
            for xp0 in Xp0:
                X,Y,T = vanderpol(mu,x0,xp0)
                plt.plot(T,X,label="mu={},␣x0={},␣x'0={}".format(mu,x0,xp0))
    plt.legend(loc = 'best')
    plt.show()

def phase(Mu,X0,Xp0):
    for mu in Mu:
        for x0 in X0:
            for xp0 in Xp0:
                X,Y,T = vanderpol(mu,x0,xp0)
                plt.plot(X,Y,label="mu={},␣x0={},␣x'0={}".format(mu,x0,xp0))
    plt.legend(loc = 'best')
    plt.show()

tracex([1],[0],[0.001,0.01,0.1])
phase([1],[0],[0.001,0.01,0.1])
```

Les figures obtenues sont données en figures 7 et 8. On s'aperçoit sur ces figures d'une stabilisation périodique, la période ne semblant pas dépendre de $x'(0)$.
Pour obtenir la dépendance vis-à-vis de μ, on passe cette fois plusieurs valeurs pour μ. On modifie la légende de sorte à ce qu'elle ne nous affiche que la valeur de μ, pour des questions d'encombrement sur le graphe (figures 9 et 10). On s'aperçoit cette fois que la période semble dépendre de μ. Plus précisément, il semblerait que la période croît avec μ.
Pour calculer la période, on part de l'observation suivante : il y a un unique maximum local sur une période. On peut donc trouver la période en considérant la durée séparant deux maxima locaux. On peut détecter un maximum local en remarquant qu'il est entouré par deux valeurs plus petites que lui.

```
def periode(mu):
    X,Y,T = vanderpol(mu,2,1)
    i = 1
```

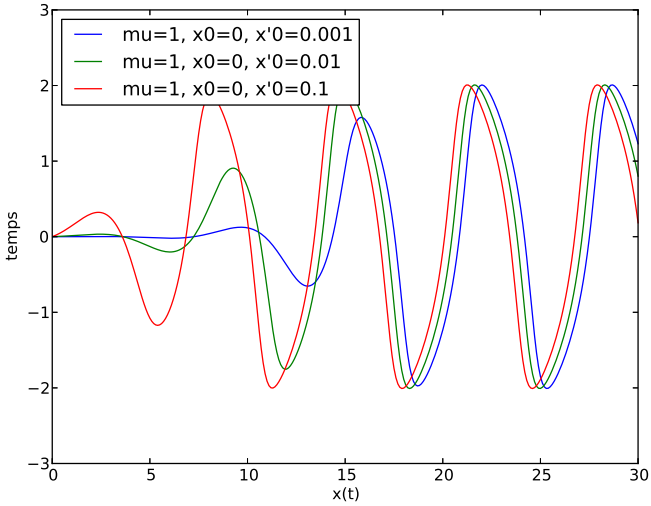

FIGURE 7 – x en fonction du temps pour différentes valeurs de $x'(O)$

```
    while X[i]< X[i-1] or X[i] < X[i+1]:
        i +=1
    t0 = T[i]
    i += 1
    while X[i]< X[i-1] or X[i] < X[i+1]:
        i += 1
    t1 = T[i]
    return t1-t0


Mu = np.linspace(0,4,41)
plt.plot(Mu,[periode(mu) for mu in Mu])
plt.xlabel('mu')
plt.ylabel('période')
plt.savefig('py044-5.eps')
plt.show()
```

Le graphe obtenu est celui de la figure 11.

Enfin, pour résoudre l'équation avec exitation, on redéfinit la fonction F donnant l'équation différentielle, en ajoutant cette fois les paramètres A et ω. On décide de partir de la position de repos ($x(0) = 0$, $x'(0) = 0$).

```
def F2(mu,A,om):
    return lambda Y,t: [Y[1],mu * (1- Y[0]**2)*Y[1] - Y[0]+ A * math.sin(2*math.pi * t * om)]

def vanderpolexite(mu,A,om):
    T = np.linspace(0,100,1001)
    Z = si.odeint(F2(mu,A,om),[0,0],T)
    X = [z[0] for z in Z]
    Y = [z[1] for z in Z]
    return X,Y,T

X,Y,T = vanderpolexite(8.53,1.2,0.1)
plt.plot(T,X)
plt.xlabel('x(t)')
```

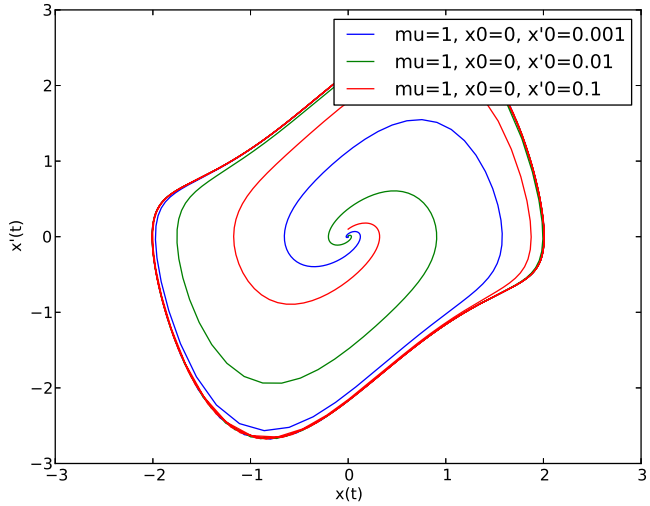

FIGURE 8 – Diagramme de phase pour différentes valeurs de $x'(0)$

```
plt.ylabel('temps')
plt.savefig('py044-6.eps')
plt.show()
```

La courbe obtenue est celle de la figure 12

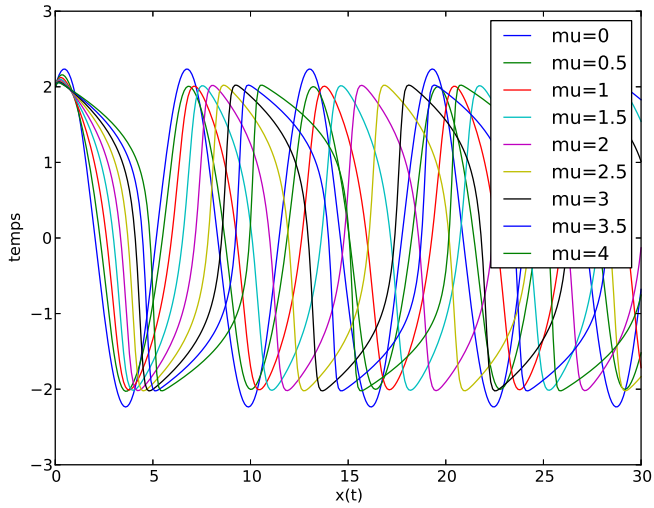

FIGURE 9 – x en fonction du temps pour différentes valeurs de μ

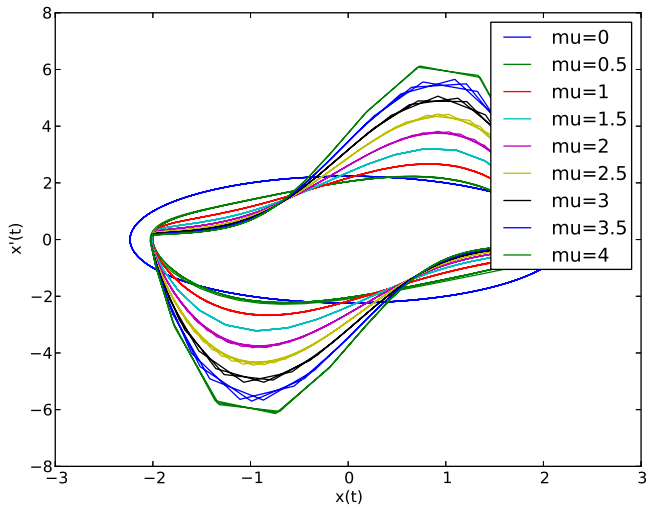

FIGURE 10 – Diagramme de phase pour différentes valeurs de μ

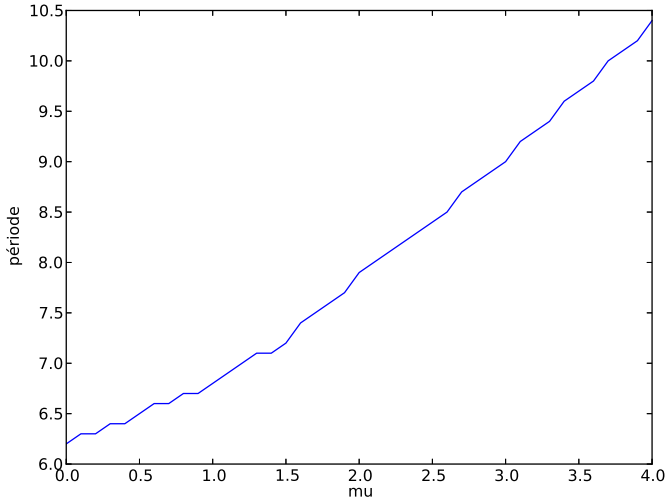

FIGURE 11 – Période en fonction de μ

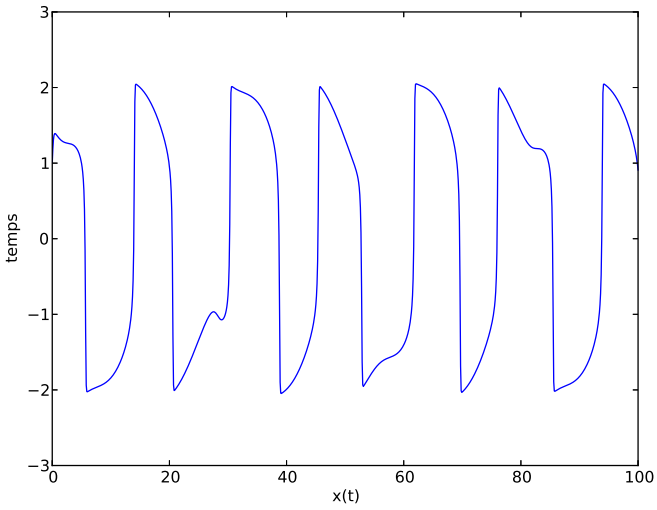

FIGURE 12 – Résolution avec exitation.