Corrigé du TP 9 Intégration - Méthode d'Euler

Chenevois-Jouhet-Junier

1 Outils

```
#imports des modules
import math,time
import numpy as np
import scipy.integrate
import matplotlib.pyplot as plt
#le module test contient les fonctions de test, il doit etre dans le meme repertoire
#son contenu est donné ci-dessous
from test import *
```

2 Contenu du module test

```
#tableau des arguments f, a, b, nom_fonction
#pour tester les méthodes d'intégration

BENCH = [(lambda t : t,0,1, 't->t'), (lambda t : t**2, 0, 1, 't->t**2'),
(lambda t : t**3, 0, 1, 't->t**3'), (lambda t:t**10,0,1, 't->t**10'),
(math.cos,0,math.pi/2,'t->cos(t)'), (math.exp,-3,3, 't->exp(t)')]


def testeur(methode, benchmark):
    '''Teste une méthode d'intégration methode(f,a,b,n) avec le tableau
    benchmark  d'arguments f, a, b, nom '''
    for args in benchmark:
        f, a, b, nom = args
        #fonction methode2 d'un seul parametre n avec f,a, b  fixés
        methode2 = lambda n : methode(f, a, b, n)
        for n in [10,10**2,10**3]:
            resint = methode2(n)
            print('Intégrale de %s sur [%.3f;%.3f]'%(nom, a, b),
            'avec %s subdivisions : %.10f'%(n, resint))
        print('Avec  scipy.integrate.quad',
        ': %.10f \n'%(scipy.integrate.quad(f, a, b)[0]))


def chrono(function, valexacte):
    def function2(*args,**kwargs):
        debut = time.perf_counter()
        res = function(*args,**kwargs)
        fin = time.perf_counter()
        return res,'%s en %s secondes avec %d décimales justes'%(res,fin-debut,
        round(math.log(abs(res-valexacte))/math.log(10)))
    return function2
```

3 Trois méthodes d'intégration numérique

3.1 Exo 1 Méthode des rectangles (1/2)

```
def rectangles(f, a, b, n):
    '''fonction prenant en entrée une fonction, deux bornes et
    un nombre de pas n, et renvoyant l'approximation Rn de
    l'intégrale de la fonction'''
    x = a
    pas = float((b-a))/n
    r = f(x)
    for i in range(1,n):
         x += pas
         r += f(x)
    return r*pas
```

3.2 Exo 2 Méthode des rectangles (2/2)

```
"""
>>> testeur(rectangles, BENCH)
Intégrale de t->t sur [0.000;1.000] avec 10 subdivisions : 0.4500000000
Intégrale de t->t sur [0.000;1.000] avec 100 subdivisions : 0.4950000000
Intégrale de t->t sur [0.000;1.000] avec 1000 subdivisions : 0.4995000000
Avec  scipy.integrate.quad : 0.5000000000

Intégrale de t->t**2 sur [0.000;1.000] avec 10 subdivisions : 0.2850000000
Intégrale de t->t**2 sur [0.000;1.000] avec 100 subdivisions : 0.3283500000
Intégrale de t->t**2 sur [0.000;1.000] avec 1000 subdivisions : 0.3328335000
Avec  scipy.integrate.quad : 0.3333333333

Intégrale de t->t**3 sur [0.000;1.000] avec 10 subdivisions : 0.2025000000
Intégrale de t->t**3 sur [0.000;1.000] avec 100 subdivisions : 0.2450250000
Intégrale de t->t**3 sur [0.000;1.000] avec 1000 subdivisions : 0.2495002500
Avec  scipy.integrate.quad : 0.2500000000

Intégrale de t->t**10 sur [0.000;1.000] avec 10 subdivisions : 0.0491434192
Intégrale de t->t**10 sur [0.000;1.000] avec 100 subdivisions : 0.0859924142
Intégrale de t->t**10 sur [0.000;1.000] avec 1000 subdivisions : 0.0904099242
Avec  scipy.integrate.quad : 0.0909090909

Intégrale de t->cos(t) sur [0.000;1.571] avec 10 subdivisions : 1.0764828027
Intégrale de t->cos(t) sur [0.000;1.571] avec 100 subdivisions : 1.0078334199
Intégrale de t->cos(t) sur [0.000;1.571] avec 1000 subdivisions : 1.0007851925
Avec  scipy.integrate.quad : 1.0000000000

Intégrale de t->exp(t) sur [-3.000;3.000] avec 10 subdivisions : 14.6225215956
Intégrale de t->exp(t) sur [-3.000;3.000] avec 100 subdivisions : 19.4406877235
Intégrale de t->exp(t) sur [-3.000;3.000] avec 1000 subdivisions : 19.9757027125
Avec  scipy.integrate.quad : 20.0357498548
"""
```

La différence avec la valeur réelle de l'intégrale semble être majorée par $\frac{1}{n}$ ce que confirme le théorème suivant si f est suffisamment dérivable.

Théorème : Soit f une fonction $\mathcal{C}^1$ sur un intervalle $[a;b]$, soit n points uniformément répartis dans $[a;b]$ avec $\sigma_k = a + k\frac{b-a}{n}$, pour $0 \leqslant k \leqslant n-1$.

Si on note $R_n^{(g)} = \frac{b-a}{n}\sum_{k=0}^{n-1} f(\sigma_k)$ la somme des aires des rectangles à gauche et $R_n^{(d)} = \frac{b-a}{n}\sum_{k=1}^{n} f(\sigma_k)$ la somme des aires des rectangles à droite et $\mu_1 = \sup_{[a;b]}|f'|$ alors :

$$\left| \int_a^b f(t)dt - R_n^{(g)} \right| \leqslant \mu_1 \frac{(b-a)^2}{2n} \qquad \text{et} \qquad \left| \int_a^b f(t)dt - R_n^{(d)} \right| \leqslant \mu_1 \frac{(b-a)^2}{2n}$$

3.3 Exo 3 Méthode des trapèzes

```
def trapeze(f,a,b,n):
    return rectangles(f,a,b,n) + (f(b)-f(a))*(b-a)/(2.*n)

"""
>>> testeur(trapeze, BENCH)
Intégrale de t->t sur [0.000;1.000] avec 10 subdivisions : 0.5000000000
Intégrale de t->t sur [0.000;1.000] avec 100 subdivisions : 0.5000000000
Intégrale de t->t sur [0.000;1.000] avec 1000 subdivisions : 0.5000000000
Avec  scipy.integrate.quad : 0.5000000000

Intégrale de t->t**2 sur [0.000;1.000] avec 10 subdivisions : 0.3350000000
Intégrale de t->t**2 sur [0.000;1.000] avec 100 subdivisions : 0.3333500000
Intégrale de t->t**2 sur [0.000;1.000] avec 1000 subdivisions : 0.3333335000
Avec  scipy.integrate.quad : 0.3333333333

Intégrale de t->t**3 sur [0.000;1.000] avec 10 subdivisions : 0.2525000000
Intégrale de t->t**3 sur [0.000;1.000] avec 100 subdivisions : 0.2500250000
Intégrale de t->t**3 sur [0.000;1.000] avec 1000 subdivisions : 0.2500002500
Avec  scipy.integrate.quad : 0.2500000000

Intégrale de t->t**10 sur [0.000;1.000] avec 10 subdivisions : 0.0991434192
Intégrale de t->t**10 sur [0.000;1.000] avec 100 subdivisions : 0.0909924142
Intégrale de t->t**10 sur [0.000;1.000] avec 1000 subdivisions : 0.0909099242
Avec  scipy.integrate.quad : 0.0909090909

Intégrale de t->cos(t) sur [0.000;1.571] avec 10 subdivisions : 0.9979429864
Intégrale de t->cos(t) sur [0.000;1.571] avec 100 subdivisions : 0.9999794382
Intégrale de t->cos(t) sur [0.000;1.571] avec 1000 subdivisions : 0.9999997944
Avec  scipy.integrate.quad : 1.0000000000

Intégrale de t->exp(t) sur [-3.000;3.000] avec 10 subdivisions : 20.6332465521
Intégrale de t->exp(t) sur [-3.000;3.000] avec 100 subdivisions : 20.0417602192
Intégrale de t->exp(t) sur [-3.000;3.000] avec 1000 subdivisions : 20.0358099620
Avec  scipy.integrate.quad : 20.0357498548
"""
```

La méthode des Trapèzes est exacte pour les fonctions affines. La différence avec la valeur réelle de l'intégrale semble être majorée par $\frac{1}{n^2}$ ce que confirme le théorème suivant si f est suffisamment dérivable.

Théorème : Soit f une fonction $\mathcal{C}^2$ sur un intervalle $[a;b]$, soit n points uniformément répartis dans $[a;b]$ avec $\sigma_k = a + k\frac{b-a}{n}$, pour $0 \leqslant k \leqslant n-1$.

Si on note $T_n = \frac{b-a}{2n} \sum_{k=0}^{n-1} f(\sigma_k) + f(\sigma_{k+1}) = \frac{R_n^{(g)} + R_n^{(d)}}{2}$ la somme des aires des trapèzes construits sur les sommets des rectangles à gauche et à droite et $\mu_2 = \sup_{[a;b]} |f''|$ alors :

$$\left| \int_a^b f(t)dt - T_n \right| \leqslant \mu_2 \frac{(b-a)^3}{12n^2}$$

3.4 Exo 4 Méthode de Simpson

```
def simpson(f,a,b,n):
```

```
    pas = float(b-a)/n
    x,y = a,a+pas
    m = (x+y)/2.
    fx,fy,fm = f(x),f(y),f(m)
    s = fx + 4*fm + fy
    for i in range(1,n):
        x, y, m = y, y+pas, m+pas
        fx, fy, fm = fy,f(y),f(m)
        s += fx + 4*fm  + fy
    return s*pas/6.

"""
>>> testeur(simpson, BENCH)
Intégrale de t->t sur [0.000;1.000] avec 10 subdivisions : 0.5000000000
Intégrale de t->t sur [0.000;1.000] avec 100 subdivisions : 0.5000000000
Intégrale de t->t sur [0.000;1.000] avec 1000 subdivisions : 0.5000000000
Avec  scipy.integrate.quad : 0.5000000000

Intégrale de t->t**2 sur [0.000;1.000] avec 10 subdivisions : 0.3333333333
Intégrale de t->t**2 sur [0.000;1.000] avec 100 subdivisions : 0.3333333333
Intégrale de t->t**2 sur [0.000;1.000] avec 1000 subdivisions : 0.3333333333
Avec  scipy.integrate.quad : 0.3333333333

Intégrale de t->t**3 sur [0.000;1.000] avec 10 subdivisions : 0.2500000000
Intégrale de t->t**3 sur [0.000;1.000] avec 100 subdivisions : 0.2500000000
Intégrale de t->t**3 sur [0.000;1.000] avec 1000 subdivisions : 0.2500000000
Avec  scipy.integrate.quad : 0.2500000000

Intégrale de t->t**10 sur [0.000;1.000] avec 10 subdivisions : 0.0909337800
Intégrale de t->t**10 sur [0.000;1.000] avec 100 subdivisions : 0.0909090934
Intégrale de t->t**10 sur [0.000;1.000] avec 1000 subdivisions : 0.0909090909
Avec  scipy.integrate.quad : 0.0909090909

Intégrale de t->cos(t) sur [0.000;1.571] avec 10 subdivisions : 1.0000002115
Intégrale de t->cos(t) sur [0.000;1.571] avec 100 subdivisions : 1.0000000000
Intégrale de t->cos(t) sur [0.000;1.571] avec 1000 subdivisions : 1.0000000000
Avec  scipy.integrate.quad : 1.0000000000

Intégrale de t->exp(t) sur [-3.000;3.000] avec 10 subdivisions : 20.0366418939
Intégrale de t->exp(t) sur [-3.000;3.000] avec 100 subdivisions : 20.0357499450
Intégrale de t->exp(t) sur [-3.000;3.000] avec 1000 subdivisions : 20.0357498548
Avec  scipy.integrate.quad : 20.0357498548
"""
```

La méthode de Simpson est exacte pour les fonctions affines (en considérant les fonctions affines comme des trinomes dégénérés) et naturellement pour les trinomes.

Elle semble être aussi exacte pour les polynômes de degré 3.

On peut démontrer, si f est $\mathcal{C}^4$, que la différence avec la valeur réelle de l'intégrale a un majorant en $\frac{1}{n^4}$.

Théorème : Soit f une fonction $\mathcal{C}^4$ sur un intervalle $[a;b]$, soit n points uniformément répartis dans $[a;b]$ avec $\sigma_k = a + k\frac{b-a}{n}$, pour $0 \leqslant k \leqslant n-1$.

Si on remplace sur chaque intervalle $[\sigma_k;\sigma_{k+1}]$ la fonction f par la fonction polynôme du second degré prenant les mêmes valeurs en σ_k, $\frac{\sigma_k+\sigma_{k+1}}{2}$ et σ_{k+1} alors la méthode de Simpson approche $\int_a^b f(t)dt$ par la somme :

$$P_n = \frac{b-a}{n}\left[\frac{1}{6}f(\sigma_0) + \frac{1}{3}\sum_{k=1}^{n-1} f(\sigma_k) + \frac{2}{3}\sum_{k=0}^{n-1} f\left(\frac{\sigma_k+\sigma_{k+1}}{2}\right) + \frac{1}{6}f(\sigma_n)\right]$$

Si on note $\mu_4 = \sup_{[a;b]}|f^{(4)}|$ alors :

$$\left| \int_a^b f(t)dt - P_n \right| \leqslant \mu_4 \frac{(b-a)^5}{2880n^4}$$

3.5 Exo5 Estimation de la complexité en temps

- Pour R_n (méthode des rectangles) : n évaluations
- Pour T_n (méthode des trapèzes) : $2n$ évaluations; $n+2$ voire $n+1$ en finassant
- Pour P_n (méthode de Simpson) : $3n$ évaluations; $2n+2$ voire $2n+1$ en finassant

Dans les trois cas, la complexité est de l'ordre de n.

3.6 Exo 6 Comparaison des méthodes d'intégration

```
def exo6():
    a, b, f = 0,math.pi/2.,math.cos
    for n in [10**2,10**4,10**6]:
        print('Pour %s subdivisions'%n)
        print('Méthode des rectangles : \n', chrono(rectangles, 1)(f,a,b,n))
        print('Méthode des trapezes : \n', chrono(trapeze, 1)(f,a,b,n))
        print('Méthode de Simpson : \n',chrono(simpson, 1)(f,a,b,n))
        print()

"""
>>> exo6()
Pour 100 subdivisions
Méthode des rectangles :
1.0078334198735823 en 5.912780761718750e-05 secondes avec 2 décimales justes
Méthode des trapezes :
0.9999794382396079 en 5.698204040527344e-05 secondes avec 5 décimales justes
Méthode de Simpson :
1.0000000000211398 en 0.0001440048217773437 5 secondes avec 11 décimales justes

Pour 10000 subdivisions
Méthode des rectangles :
1.0000785377600803 en 0.003718137741088867 secondes avec 4 décimales justes
Méthode des trapezes :
0.9999999979437405 en 0.0032684803009033203 secondes avec 9 décimales justes
Méthode de Simpson :
0.9999999999999089 en 0.008365869522094727 secondes avec 13 décimales justes

Pour 1000000 subdivisions
Méthode des rectangles :
1.0000007854017243 en 0.2927272319793701 secondes avec 6 décimales justes
Méthode des trapezes :
1.000000000003561 en 0.28957509994506836 secondes avec 11 décimales justes
Méthode de Simpson :
1.000000000003723 en 0.7962665557861328 secondes avec 11 décimales justes
"""
```

Notez que les problèmes d'approximation des réels par des flottants avec une précision d'environ 16 décimales font que la différence de précision entre la méthode de Simpson et la méthode des trapèzes n'est plus visible lorsque le nombre de subdivisions est assez grand.

4 Méthode d'Euler pour l'intégration numérique d'une équation différentielle

4.1 Exo 7 Schéma d'Euler pour intégrer $y' = F(t, y)$

```
def euler(F,a,y0,b,n):
    '''renvoie les n+1 valeurs approchées de y solution de y'(t)=F(t,y(t))
    et y(a)=y0 aux temps tk=a+k(b-a)/n
    '''
    les_yk = [y0] #initialisation de la valeur
    t, h = a, float(b-a)/n #initialisation du temps et du pas
    for i in range(n):
        yk = les_yk[-1]
        les_yk.append(yk + F(t,yk)*h) #mise à jour de la liste de valeurs
        t += h  #mise à jour du temps
    return les_yk
```

4.2 Exo 8 Schéma d'Euler pour la fonction exponentielle sur $[0; 1]$

```
def f0(t,y):
    return y
```

Approximation de la fonction exponentielle par la méthode d'Euler sur [0;1].

```
"""
>>> euler(f0,0,1,1,10)
[1, 1.1, 1.2100000000000002, 1.3310000000000002, 1.4641000000000002, 1.61051,
1.7715610000000002, 1.9487171, 2.1435888100000002, 2.357947691, 2.5937424601]
"""

def exo8():
    '''représentation graphique de l'approximation de la fonction exponentielle
    par la méthode d'Euler sur $[0;1]$
    '''
    #tracé de l'approximation
    x = np.linspace(0,1,11) #10 subdivision donc 11 bornes
    y = euler(f0,0,1,1,10)
    plt.plot(x,y,'-o',color='red')
    #tracé de la courbe de la fonction exponentielle
    x = np.linspace(0,1,1000)
    y = np.exp(x)
    plt.plot(x,y,'b-')
    plt.legend(['Euler','Solution exacte'], loc='upper left')
    plt.title("Méthode d'Euler pour y'=y")
    plt.savefig('expo10.pdf')
    plt.show()
    #plt.clf()
```

4.3 Exo 9

```
def f3(t, y):
    [a,b] = y
    return np.array([b, -a])


"""
```

```
>>> Y100 = euler(f3,0,np.array([0,1]),np.pi*4,100)
>>> Y100[:3]
[array([0, 1]), array([ 0.12566371,  1.   ]), array([ 0.25132741,  0.98420863])]
"""
```

4.4 Exo 10 Schéma d'Euler vectorialisé pour intégrer $y'' = F(t, y, y')$

```
def exo10():
    '''représentation graphique de l'approximation de la fonction sinus
    solution de y"=-y et y(0)=0 sur [0;4*pi] par la méthode d'Euler
    Théoriquement la courbe paramétrée par (y, y') = (sin, cos) est un cercle
    '''
    #solution approchée par la méthode d'Euler
    Y100 = np.array(euler(f3,0,np.array([0,1]),np.pi*4,100))
    #premier graphique : courbe approchée
    plt.subplot(211)
    #tracé de l'approximation
    x = np.linspace(0,4*np.pi,101)
    y = Y100[:,0]
    plt.plot(x,y,'-o',color='red')
    #tracé de la courbe de la fonction sinus
    x = np.linspace(0,4*np.pi,1000)
    y = np.sin(x)
    plt.plot(x,y,'b-')
    plt.legend(['Euler','Solution exacte'], loc='upper left')
    plt.title(r"Méthode d'Euler pour $y''=-y$ avec n=100")
    #deuxième graphique : portraits de phases des solutions approchées
    #pour des découpages en n=100, n=10*3 ou n=10**4 sous-intervalles
    #c'est la courbe paramétrée reliant les (yk,y'k)
    #comme y''=-y et y(0)=1 le portrait de phase de la solution exacte
    #sur [0;4*pi] devrait etre un cercle de rayon 1
    plt.subplot(212)
    #pour tracer les axes
    plt.axhline(color='black')
    plt.axvline(color='black')
    for (n,col,style, lab) in [(100,'red','-', r'$n=100$'),
    (10**3,'blue','--', r'$n=10^3$'),(10**4,'green','-.', r'$n=10^4$')]:
        Y = np.array(euler(f3,0,np.array([0,1]),np.pi*4,n))
        x,y = Y[:,0],Y[:,1]
        plt.plot(x,y,style,color=col, label=lab)
    plt.legend(loc='upper left')
    plt.title("Portrait de phase pour $y''=-y$")
    plt.savefig('sinus10.pdf')
    plt.show()
```

5 Extrait du sujet posé au concours Centrale 2015

Soit y une fonction de classe $\mathcal{C}^2$ sur $\mathbb{R}$ et $t_{\min}$ et $t_{\max}$ deux réels tels que $t_{\min} < t_{\max}$. On note I l'intervalle $[t_{\min}\,,\, t_{\max}]$. On considère une équation différentielle $\mathcal{E}_1$ du second ordre de la forme:

$$\forall t \in I, \qquad y''(t) = f(y(t))$$

où f est une fonction donnée, continue sur $\mathbb{R}$.

Les conditions initiales sont $y_0 = y\,(t_{\min})$ et $z_0 = y'\,(t_{\min})$. On suppose que le système physique étudié est conservatif : il existe une quantité indépendante du temps (énergie, quantité de mouvement, ...), notée E , qui vérifie l'équation où $g' = -f$.

$$\forall t \in I, \qquad \frac{1}{2}y'(t)^2 + g(y(t)) = E$$

On introduit la fonction $z : I \to \mathbb{R}$ définie par $\forall t \in I$, $z(t) = y'(t)$.

5.1 Exo 11 Mise en forme du problème

- Question 1 :

  L'équation $\mathcal{E}_1$ équivaut au système différentiel du premier ordre : $(S)\begin{cases} y'(t) = z(t) \\ z'(t) = f(y(t)) \end{cases}$.

- Question 2 :

  Soit n un entier strictement supérieur à 1 et $J_n = [[0\,,\, n-1]]$.

  On pose $h = \frac{t_{\max} - t_{\min}}{n-1}$ et $\forall i \in J_n$, $t_i = t_{\min} + ih$. Pour tout entier $0 \leqslant i \leqslant n-2$ on a :

$$y(t_{i+1}) - y(t_i) = \int_{t_i}^{t_{i+1}} y'(t)\,\mathrm{d}t = \int_{t_i}^{t_{i+1}} z(t)\,\mathrm{d}t$$
$$z(t_{i+1}) - z(t_i) = \int_{t_i}^{t_{i+1}} z'(t)\,\mathrm{d}t = \int_{t_i}^{t_{i+1}} y''(t)\,\mathrm{d}t = \int_{t_i}^{t_{i+1}} f(y(t))\,\mathrm{d}t$$

5.2 Exo 12 Schéma d'Euler explicite

- Question 1 :

  Les équations ci-dessus permettent de définir deux suites $(y_i)_{i\in J_n}$ et $(z_i)_{i\in J_n}$ où y_i et z_i sont des valeurs approchées de $y(t_i)$ et $z(t_i)$.

  $\begin{cases} y_0 = y(t_{min}) \\ y_{i+1} = y_i + hz_i \end{cases}$ et $\begin{cases} z_0 = y'(t_{min}) \\ z_{i+1} = z_i + hf(y_i) \end{cases}$

- Question 2 :

```
def euler(f, n , tmin, tmax, ytmin, yptmin):
    h  = (tmax - tmin)/(n - 1)
    ypreced = ytmin
    zpreced = yptmin
    yy = [ypreced]
    zz = [zpreced]
    for k in range(n - 1):
        (ynouveau, znouveau) = (ypreced + h*zpreced, zpreced + h*f(ypreced))
        yy.append(ynouveau)
        zz.append(znouveau)
        (ypreced, zpreced) = (ynouveau, znouveau)
    return (yy, zz)
```

- Question 3
  - Question 3 a) :

    En intégrant l'équation différentielle $y''(t) = -\omega^2 y(t)$ on obtient une équation de conservation :

$$(y'(t))^2 + \omega^2 y^2(t) = E = (y'(0))^2 + \omega^2 y^2(0)$$

  - Question 3 b) :

    On note E_i la valeur approchée de E à l'instant t_i, $i \in J_n$, calculée en utilisant les valeurs approchées de $y(t_i)$ et $z(t_i)$ obtenues à la question 1..

$$E_{i+1} - E_i = z_{i+1}^2 + \omega^2 y_{i+1}^2 - z_i^2 - \omega^2 y_i^2$$
$$E_{i+1} - E_i = (z_i - h\omega^2 y_i)^2 + \omega^2 (y_i + h z_i)^2 - z_i^2 - \omega^2 y_i^2$$
$$E_{i+1} - E_i = -2h z_i \omega^2 y_i + h^2 \omega^4 y_i^2 + 2\omega^2 h y_i z_i + \omega^2 h^2 z_i^2$$
$$E_{i+1} - E_i = h^2 \omega^2 (z_i^2 + \omega^2 y_i^2)$$
$$E_{i+1} - E_i = h^2 \omega^2 E_i^2$$

– Question 3 c) :

Un schéma numérique qui satisfait à la conservation de E serait tel que pour tout entier $0 \leqslant i \leqslant n-2\,, E_{i+1} - E_i = 0$.

Pour le schéma d'Euler explicite, c'est impossible car on aurait pour tout $i \in J_n$, $E_i = 0$ d'après l'égalité précédente c'est-à-dire $y_i = 0$ et $z_i = 0$, ce qui est absurde.

– Question 3 d) :

En portant les valeurs de y_i sur l'axe des abscisses et de z_i sur l'axe des ordonnées, le graphe d'un schéma respectant la conservation de E serait tel que pour tout entier $i \in J_n$, $z_i^2 + \omega^2 y_i^2 = E$. On reconnaît l'équation d'une ellipse.

– Question 3 e) :

Des relations de récurrence $\begin{cases} y_0 = y(t_{min}) \\ y_{i+1} = y_i + h z_i \end{cases}$ et $\begin{cases} z_0 = y'(t_{min}) \\ z_{i+1} = z_i + h f(y_i) \end{cases}$, avec $f(y_i) = -\omega^2 y_i$, on déduit que $y_i > 0 \Leftrightarrow z_{i+1} < z_i$ et $z_i > 0 \Leftrightarrow y_{i+1} > y_i$.

De plus lorsque $y'(t_i) \approx z_i$ est positif alors $y_i \approx y(t_i)$ augmente et lorsque $y'(t_i) \approx z_i$ est négatif alors $y_i \approx y(t_i)$ diminue.

On en déduit que le nuage de points de coordonnées $(x_i\,, y_i)_{i \in J_n}$ a l'allure d'une **spirale**.

Par ailleurs pour tout entier $i \in J_n$, $E_{i+1} - E_i = h^2 \omega^2 E_i^2$ avec $E_i > 0$ par conséquent la suite (E_i) est strictement croissante ainsi la suite $(z_i^2 + \omega^2 y_i^2)$ est strictement croissante ce qui se traduit par une divergence de la spirale (la distance du point de coordonnées $(\omega y_i, z_i)$ à l'origine tend vers $+\infty$ donc celle de (y_i, z_i) à l'origine tend vers $+\infty$ également puisque la multiplication par ω de y_i correspond simplement à une dilatation selon l'axe des abscisses).

5.3 Exo 13 Schéma d'intégration de Verlet

- Question 1 :

Pour tout entier $i \in J_n$ on a :

$$y_{i+1} = y_i + h z_i + \frac{h^2}{2} f_i \text{ avec } f_i = f(y_i)$$
$$y_{i+1} = y_i (1 - \frac{\omega^2 h^2}{2}) + h z_i$$
$$z_{i+1} = z_i + \frac{h}{2} (f_i + f_{i+1}) \text{ avec } f_{i+1} = f(y_{i+1})$$
$$z_{i+1} = z_i - \frac{h\omega^2}{2} (y_i + y_{i+1})$$
$$z_{i+1} = z_i - \frac{h\omega^2}{2} (y_i (2 - \frac{\omega^2 h^2}{2}) + h z_i)$$

```
def verlet(f, n , tmin, tmax, ytmin, yptmin):
    h  = (tmax - tmin)/(n - 1)
    ypreced = ytmin
    zpreced = yptmin
    fpreced = f(ypreced)
    yy = [ypreced]
    zz = [zpreced]
    for k in range(n - 1):
        ynouveau = ypreced + h*zpreced + h**2/2*fpreced
```

```
        fnouveau = f(ynouveau)
        znouveau = zpreced + h/2*(fnouveau + fpreced)
        yy.append(ynouveau)
        zz.append(znouveau)
        (ypreced, zpreced, fpreced) = (ynouveau, znouveau, fnouveau)
    return (yy, zz)
"""
In [4]: yy, zz = verlet(lambda y : -(2*np.pi)**2*y, 100, 0, 3, 3, 0)

In [5]: yy[-3:], zz[-3:]
Out[5]:
([2.8152395363671467, 2.9606719372849324, 2.9987740873949714],
 [6.483226837453586, 3.028320091959096, -0.53636928460061651])
"""
```

- Question 2 :
  - Question 2 a) : pour tout entier $i \in J_n$ on a :

$$
\begin{aligned}
E_{i+1} - E_i &= z_{i+1}^2 + \omega^2 y_{i+1}^2 - z_i^2 - \omega^2 y_i^2 \\
E_{i+1} - E_i &= \left(z_i - \frac{h\omega^2}{2}(y_i(2 - \frac{\omega^2 h^2}{2}) + hz_i) \right)^2 + \omega^2 \left(y_i(1 - \frac{\omega^2 h^2}{2}) + hz_i \right)^2 - z_i^2 - \omega^2 y_i^2 \\
E_{i+1} - E_i &= \left(z_i(1 - \frac{h^2\omega^2}{2}) + y_i(\frac{\omega^4 h^3}{4} - h\omega^2) \right)^2 + \omega^2 \left(y_i(1 - \frac{\omega^2 h^2}{2}) + hz_i \right)^2 - z_i^2 - \omega^2 y_i^2 \\
E_{i+1} - E_i &= -\frac{1}{4} y_i^2 h^4 \omega^6 + y_i^2 h^6 \frac{\omega^8}{16} + z_i^2 h^4 \frac{\omega^4}{4} + 2 z_i y_i (1 - \frac{h^2\omega^2}{2}) \frac{\omega^4 h^3}{4}
\end{aligned}
$$

    On en déduit que pour le schéma de Verlet on a pour tout $i \in J_n$, $E_{i+1} - E_i = O(h^3)$.

  - Question 2 b) :

    Le nuage de points de coordonnées $(x_i\,, y_i)_{i \in J_n}$ obtenu avec le schéma numérique de Verlet est beaucoup plus proche de l'ellipse attendue, d'équation $(y'(t))^2 + \omega^2 y^2(t) = E$, qui traduit la conservation de l'énergie pour la solution théorique de l'équation différentielle $\forall t \in I,\ y''(t) = -\omega^2 y(t)$.

  - Question 2 c) :

    On en déduit que le schéma de Verlet traduit mieux la conservation de l'énergie que le schéma d'Euler (car pour tout $i \in J_n$, $E_{i+1} - E_i = O(h^3)$) et que c'est un schéma d'intégration plus précis.

6 Pour ceux qui s'ennuient

6.1 Exo 14

Résolution de $y'' = y' + 6y$ avec les conditions initiales $y(0) = -1$ et $y'(0) = 2$.

Les solutions de l'équation sans conditions initiales sont les $t \mapsto K_1 \mathrm{e}^{-2t} + K_2 \mathrm{e}^{3t}$. L'unique solution avec les conditions initiales est l'application $t \mapsto -\mathrm{e}^{-2t}$.

```
def f4(t, y):
    [a,b] = y
    return np.array([b, b + 6*a])

def exo14():
    t1 = np.linspace(0, 10, 1001)
    Y1 = euler(f4, 0, np.array([-1,2]), 10, 1000)
    y1 = np.array(Y1)[:, 0]
```

```
    plt.plot(t1, y1)
    plt.axhline(color='black')
    plt.savefig('exo11-converge.pdf')
    #plt.show()
    plt.clf()

    t2 = np.linspace(0, 20, 2001)
    Y2 = euler(f4, 0, np.array([-1,2]), 20, 2000)
    y2 = np.array(Y2)[:, 0]

    plt.plot(t2, y2)
    plt.axhline(color='black')
    plt.savefig('exo11-diverge.pdf')
    #plt.show()
    plt.clf()
    return Y1, Y2
```

On récupère en Y1 le tableau des valeurs approchées de Y pour le schéma d'Euler sur [0;10] avec $n = 1000$ et en Y2 le tableau des valeurs approchées de y pour le schéma d'Euler sur [0;20] avec $n = 2000$.

```
"""
>>> Y1,Y2 = exo14()
"""
```

Pour les deux schémas, les valeurs initiales et le pas 10/1000 sont les memes donc les valeurs approchées sont les memes pour les 1000 premières valeurs.

```
"""
>>> Y1[999:]
[array([  7.12899523e-06,   2.13955722e-05]), array([  7.34295095e-06,
2.20372677e-05])]
>>> Y2[999:1001]
[array([  7.12899523e-06,   2.13955722e-05]), array([  7.34295095e-06,
2.20372677e-05])]
"""
```

Mais au-delà de 1 (la millième valeur) les composantes de $Y(k) = \begin{pmatrix} y(k) \\ y'(k) \end{pmatrix}$ sont strictement positives.

Avec $Y(k+1) = Y(k) + h \times \begin{pmatrix} y'(k) \\ y'(k) + 6 \times y(k) \end{pmatrix}$ entre $k = 1000$ et $k = 2000$, les composantes de $Y(k)$ sont deux suites strictement croissantes et tendant vers $+\infty$.

```
"""
>>> Y2[2000]
array([  5.04887779e+07,   1.51466334e+08])
"""
```

Plus on s'éloigne de la valeur initiale exacte, plus l'accumulation des erreurs dues aux approximations successives est importante.

6.2 Exo 15

```
def f5(t, y):
    return -3*y

def exo15():
    tmax = 3.
    les_n = [30, 5, 4, 3]
    for n in les_n:
        t = np.linspace(0, tmax, 1+n)
        y = euler(f5, 0, 1, tmax, n)
```

```
        plt.plot(t, y, label='%s'%n)

    plt.axhline(color='black')
    plt.title(r"Schéma d'Euler pour $y'=-3y$ et $y(0)=1$")
    plt.legend(loc='lower left')
    plt.savefig('exo12.pdf')
    #plt.show()
    plt.clf()
```

Pour $h = \frac{t_{max}}{n} > \frac{2}{3}$, la solution «diverge» : la distance à la solution réelle tend vers l'infini quand t_{max} en fait autant. Le lecteur pourra calculer à la main les différents termes de la méthode d'Euler et comprendra ainsi le seuil des $\frac{2}{3}$.

6.3 Exo 16

```
from scipy.optimize import fsolve

def EulerRetro(F, a, y0, b, n):
    y = y0
    t = a
    h = float(b - a)/n
    les_y = [y0]
    for k in range(n):
        yk = les_y[-1]
        y = fsolve(lambda z:z-yk-h*F(t+h,z),yk)
        t += h
        les_y.append(y)
    return les_y


def exo16():
    Y = EulerRetro(lambda t,y:-3*y, 0, 1, 3, 3)
    t = np.linspace(0, 3, 4)
    plt.plot(t,Y,'o-')

    Y = EulerRetro(lambda t,y:-3*y, 0, 1, 3, 4)
    t = np.linspace(0, 3, 5)
    plt.plot(t,Y,linewidth=3)

    Y = EulerRetro(lambda t,y:-3*y, 0, 1, 3, 10)
    t = np.linspace(0, 3, 11)
    plt.plot(t,Y,'o-')

    t = np.linspace(0, 3, 100)
    f = lambda x : np.exp(-3*x)
    y = f(t)
    plt.plot(t, y)
    plt.legend([r'$h=1$',r'$h=0.75$',r'$h=0.3$', r'$y=\exp(-3x)$'])
    plt.title(r"Euler Rétrograde")

    plt.savefig('exo13-euler-retro.pdf')
    #plt.show()
    plt.clf()
```