Méthode de Bulirsch-Stoer

1. Introduction

La méthode de Bulirsch-Stoer ([1]) est une méthode d'intégration numérique des équations différentielle de la forme :

$$Y'(t) = f(Y, t) \tag{1}$$

Elle est particulièrement recommandée lorsqu'une grande précision est nécessaire, et lorsque l'évaluation de la fonction f est longue. La simulation du mouvement des planètes du système solaire est un exemple d'application de cette méthode.

Les conventions de notation et d'implémentation adoptées dans ce document sont identiques à celles du document Intégration des équations du mouvement : méthodes d'Euler.

On note H le pas de temps. Supposons que l'approximation Y_n à l'instant $t_n = nH$ soit calculée. L'approximation Y_{n+1} à l'instant $t_{n+1} = t_n + H$ est obtenue en appliquant la méthode du point milieu (méthode d'ordre 2) avec un pas

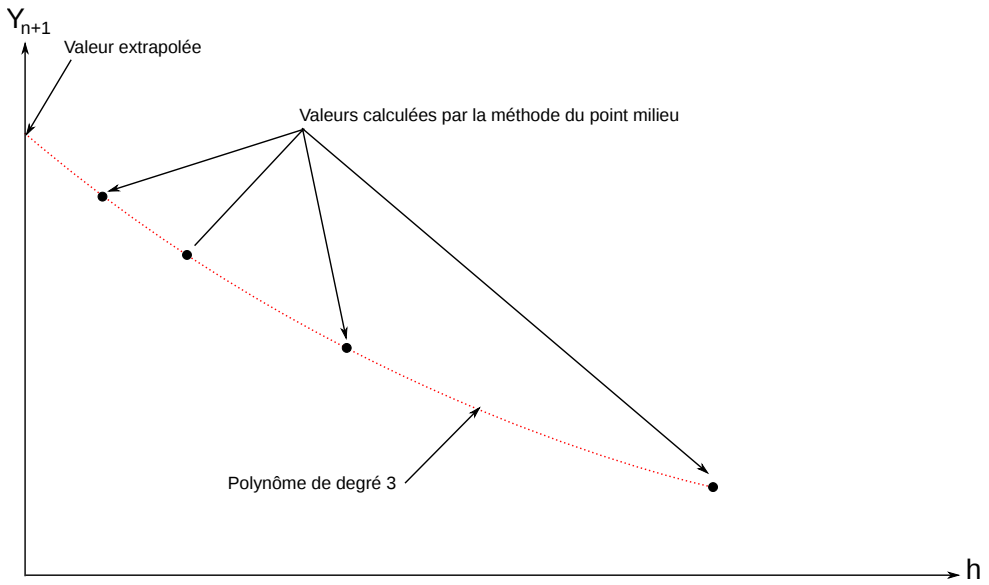
$$h = \frac{H}{m} \tag{2}$$

où m est un entier. Notons $Y_{n+1}^{(}m)$ cette approximation. La méthode consiste à calculer cette approximation pour une suite de valeurs de m croissante. La convergence de la méthode du point milieu nous permet de rechercher une approximation idéale :

$$Y_{n+1} = \lim_{m\to\infty} Y_{n+1}^{(}m) \tag{3}$$

Si l'on suppose que toutes les valeurs précédentes sont exactes, cette limite donne la valeur exacte de $Y(t_{n+1})$.

La méthode de Bulirsch-Stoer consiste à effectuer une extrapolation en $h = 0$ des valeurs de $Y_{n+1}^{(}m)$ obtenues pour quelques valeurs de h (le plus souvent moins d'une dizaine). La méthode d'extrapolation la plus simple, que nous allons utiliser, est l'extrapolation polynomiale, qui fait appel au polynôme de Lagrange passant par les points calculés. La figure suivante montre le principe de cette extrapolation.



On commence par définir la méthode du point milieu avant d'aborder la méthode d'extrapolation polynomiale.

2. Méthode du point milieu

Considérons la méthode d'Euler pour un pas h :

$$Y_{n+1} = Y_n + hf(Y_n, t_n) \tag{4}$$

La méthode du point milieu consiste à introduire le point d'abscisse $t_n + h/2$ et à évaluer la dérivée en ce point en utilisant la formule d'Euler pour un pas $h/2$. Cette dérivée est ensuite utilisée pour calculer Y_{n+1} (au lieu de la dérivée en t_n). Le schéma complet pour un pas s'écrit :

$$k_1 = hf(Y_n, t_n) \tag{5}$$

$$k_2 = hf(Y_n + \frac{1}{2}k_1, t_n + \frac{1}{2}h) \tag{6}$$

$$Y_{n+1} = Y_n + k_2 \tag{7}$$

k_2 est la dérivée évaluée au point milieu multipliée par h. Comparée à la méthode d'Euler, la méthode du point milieu a l'avantage d'être d'ordre 2 (ordre 1 pour Euler), ce qui signifie que l'erreur locale introduite par le schéma est $O(h^3)$.

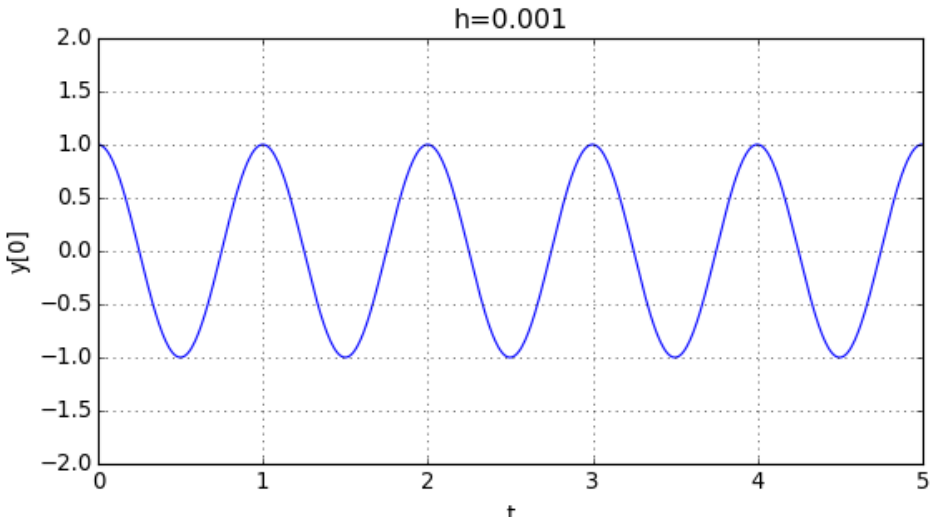
La méthode du point milieu fait partie de la classe des méthodes de Runge-Kutta, qui consistent à effectuer des évaluations des dérivées sur des points intermédiaires entre t_n et t_{n+1}. Les méthodes de Runge-Kutta sont intéressantes en elles-mêmes, surtout les méthodes d'ordre 4 ou plus. Dans le cas présent, on se limite à une méthode d'ordre 2, qui nécessite seulement deux évaluations de f par pas.

Voici une implémentation de cette méthode pour un système différentiel.

```
import numpy

w2 = (2*numpy.pi)**2
def oscillateur(Y,t):
    return numpy.array([Y[1],-w2*Y[0]])

def pas_pointmilieu(systeme,h,t,Yn):
    k1 = h*systeme(Yn,t)
    k2 = h*systeme(Yn+0.5*k1,t+0.5*h)
    return Yn+k2

def pointmilieu(systeme,Ti,T,h):
    Y = Yi
    t = 0.0
    liste_y = [Y]
    liste_t = [t]
    while t<T:
        Y = pas_pointmilieu(systeme,h,t,Y)
        t += h
        liste_y.append(Y)
        liste_t.append(t)
    return (numpy.array(liste_t),numpy.array(liste_y))
```

Voici un exemple avec l'équation de l'oscillateur harmonique :

```
from matplotlib.pyplot import *
T = 5.0
h = 1.0e-2
Yi = [1.0,0]
(t,tab_y) = pointmilieu(oscillateur,Yi,T,h)
yt = tab_y.transpose()
x = yt[0]
figure(figsize=(8,4))
plot(t,x)
xlabel('t')
ylabel('y[0]')
axis([0,T,-2,2])
title('h=0.001')
grid()
```



3. Méthode du point milieu modifiée

Soit H le pas de temps, que l'on subdivise en m intervalles de longueur

$$h = \frac{H}{m} \tag{8}$$

où m est un entier pair.

Supposons que Y_n soit calculé. On cherche Y_{n+1}, une approximation de $y(t_{n+1})$ pour $t = t_{n+1} = t_n + H$. On procède pour cela avec la méthode du point milieu appliquée successivement avec le pas $2h$:

$$z_0 = Y_n \tag{9}$$
$$z_1 = z_0 + hf(z_0, t_n) \tag{10}$$
$$z_{k+1} = z_{k-1} + 2hf(z_k, t_n + kh) \text{ pour } k = 1, 2, \ldots, m-1 \tag{11}$$
$$Y_{n+1}^{(}m) = \frac{1}{2}\left(z_m + z_{m-1} + hf(z_m, t_n + H)\right) \tag{12}$$

Le démarrage et la terminaison sont différents, c'est pourquoi on l'appelle méthode du point milieu modifiée. Une propriété importante de cette méthode est de donner une erreur qui s'exprime en fonction des puissances paires de h :

$$Y_n - y(t_n) = \sum_{i=1}^{\infty} \alpha_i h^{2i} \tag{13}$$

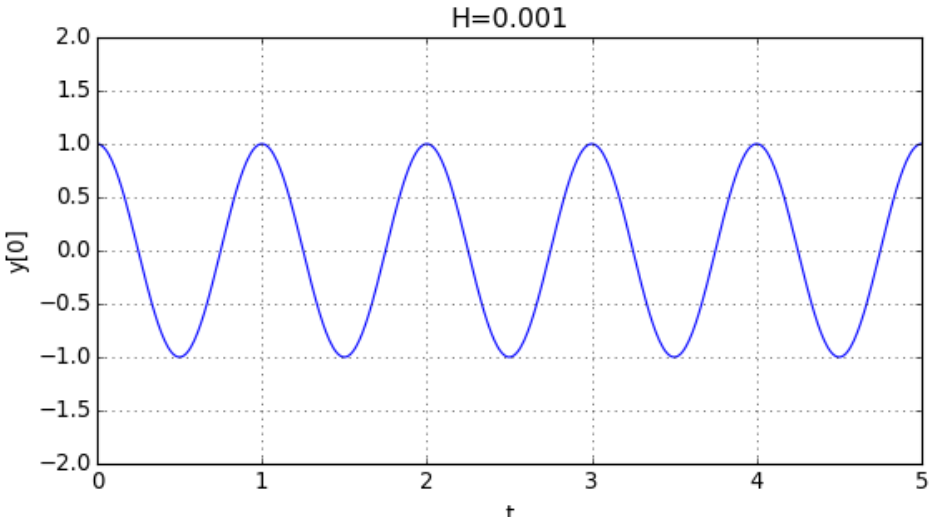
Voici une implémentation :

```
def pas_pointmilieu_modifie(systeme,H,t,Yn,m):
    h = H/m
    u = Yn
    v = u+h*systeme(u,t)
    for k in range(1,m):
        w = u+2*h*systeme(v,t+k*h)
        u = v
```

```
        v = w
    return 0.5*(v+u+h*systeme(v,t+H))


def pointmilieu_modifie(systeme,Yi,T,H,m):
    Y = Yi
    t = 0.0
    liste_y = [Y]
    liste_t = [t]
    while t<T:
        Y = pas_pointmilieu_modifie(systeme,H,t,Y,m)
        t += H
        liste_y.append(Y)
        liste_t.append(t)
    return (numpy.array(liste_t),numpy.array(liste_y))
```

Voici un test avec l'équation de l'oscillateur harmonique, pour $m = 4$:

```
T = 5.0
H = 1.0e-2
m=4
Yi = [1.0,0]
(t,tab_y) = pointmilieu_modifie(oscillateur,Yi,T,H,m)
yt = tab_y.transpose()
x = yt[0]
figure(figsize=(8,4))
plot(t,x)
xlabel('t')
ylabel('y[0]')
axis([0,T,-2,2])
title('H=0.001')
grid()
```



4. Méthode de Bulirsch-Stoer

4.a. Extrapolation polynomiale

Pour un pas H, on calcule $Y_{n+1}^{(}m)$ pour des valeurs croissantes de m, par exemple pour la suite de valeurs suivante :

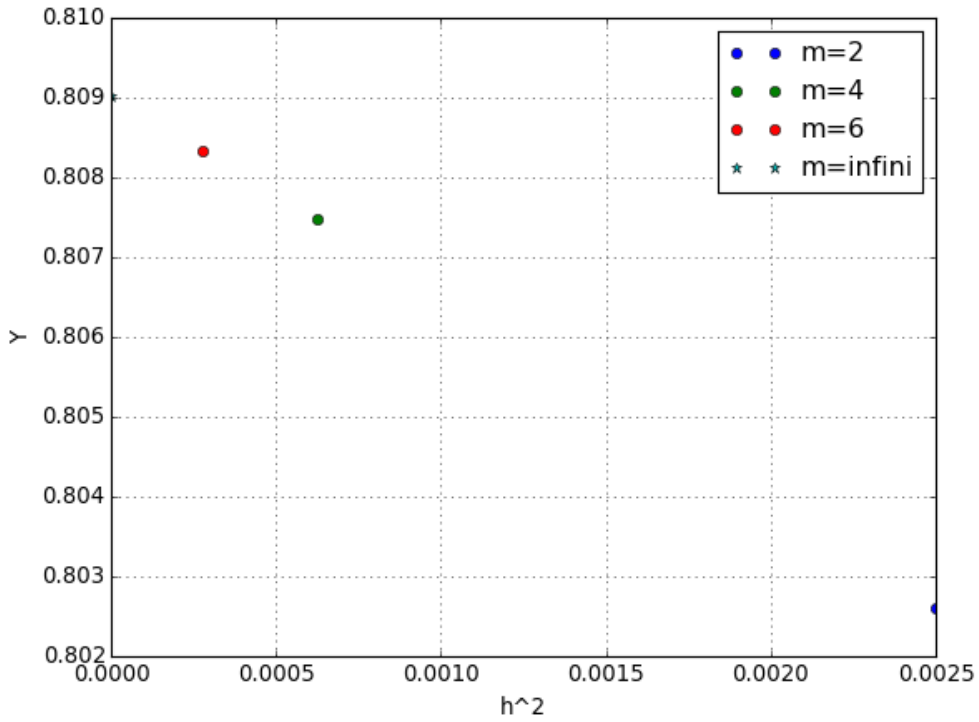
$$m_j = 2(j+1) = 2, 4, 6, 8, 10, 12, 14, \ldots \tag{14}$$

On obtient ainsi une évaluation $Y_{n+1}^{(}m)$ de plus en plus précise, pour des pas décroissants :

$$h_j = \frac{H}{m_j} \tag{15}$$

Pour illuster la méthode, considérons le premier point ($t = H$) de l'équation de l'oscillateur harmonique.

```
H = 1.0e-1
Yi = [1.0,0]
m=2
h=H/m
Y = pas_pointmilieu_modifie(oscillateur,H,0,Yi,m)
P0 = [h**2,Y[0]]
m=4
h=H/m
Y = pas_pointmilieu_modifie(oscillateur,H,0,Yi,m)
P1 = [h**2,Y[0]]
m=6
h=H/m
Y = pas_pointmilieu_modifie(oscillateur,H,0,Yi,m)
P2 = [h**2,Y[0]]
figure()
plot([P0[0]],[P0[1]],'o',label="m=2")
plot([P1[0]],[P1[1]],'o',label="m=4")
plot([P2[0]],[P2[1]],'o',label="m=6")
plot([0],[numpy.cos(2*numpy.pi*H)],'*',label='m=infini') # valeur exacte
xlabel('h^2')
ylabel('Y')
legend(loc='upper right')
grid()
```



On a tracé en abscisse le carré du pas h, car on sait que l'erreur s'exprime en fonction de ce carré.

La méthode de Bulirsch-Stoer consiste à rechercher une valeur très proche de la valeur exacte en faisant une extrapolation à $h = 0$. On considère ici le cas d'une extrapolation par une fonction polynomiale.

Dans l'exemple ci-dessus, l'extrapolation obtenue avec les deux points $m = 2$ et $m = 4$ (par un polynôme de degré 1) donne déjà une valeur très proche de la valeur exacte ($m = infini$). L'extrapolation avec les trois points $m = 2, 4, 6$ (polynôme de degré 2) donne une valeur encore plus proche de la valeur exacte. La comparasion entre ces deux extrapolations, celle pour deux points et celle pour trois points, permet d'estimer la précision du résultat.

Pour chaque valeur $m = m_j$, on considère le polynôme de Lagrange qui passe par les points déjà calculés $(h_i^2, Y_{n+1}^{(}m_i))$ pour $i = 0, 1, ..., j$. On utilise pour cela l'algorithme de Neville, détaillé dans Interpolation polynomiale : algorithme de Neville.

Si l'on reprend la notation consistant à reporter en indice les points par lequel le polynôme passe, on a :

$$\begin{array}{llll} P_0 & & & \\ P_1 & P_{01} & & \\ P_2 & P_{12} & P_{012} & \\ P_3 & P_{23} & P_{123} & P_{0123} \end{array} \tag{16}$$

Chaque ligne est calculée avec la formule de récurrence de Neville, après que la méthode du point milieu modifiée ait fourni le premier élément P_j, qui correspond au pas $h = h_j$. Pour faciliter l'implémentation, on utilise une notation à deux indices :

$$\begin{array}{cccc} A_{00} & & & \\ A_{10} & A_{11} & & \\ A_{20} & A_{21} & A_{22} & \\ A_{30} & A_{31} & A_{32} & A_{33} \end{array} \tag{17}$$

On commence par $m = m_0 = 2$. Soit $A_{00} = Y_{n+1}^{(}m_0)$ la valeur approchée obtenue à l'instant t_{n+1} avec le pas $h_0 = H/m_0$. On calcule ensuite $A_{10} = P_{01} = Y_{n+1}^{(}m_1)$, valeur approchée pour un pas $h_1 = H/m_1$. Le polynôme de degré 1 obtenu avec les deux premiers points (h_0^2, A_{00}) et (h_1^2, A_{10}) est évalué en $h^2 = 0$. Si l'on note $x = h^2$ la variable, l'application de la formule de récurrence de Neville fournit pour ce polynôme de degré 1 évalué en $x = 0$:

$$A_{11} = \frac{-h_0^2 A_{10} + h_1^2 A_{00}}{h_1^2 - h_0^2} = A_{10} + \frac{A_{10} - A_{00}}{\left(\frac{h_0}{h_1}\right)^2 - 1} \tag{18}$$

On remarque que la notation avec indices des points est P_{01}. Avec cette notation, on doit utiliser les indices extrêmes, ici 0 et 1, pour savoir quelles h_i^2 reporter dans la formule de récurrence. La seconde écriture fait clairement apparaître la correction apportée à la première estimation A_{10}. Cette correction permet d'estimer l'erreur.

Si à ce stade l'estimation de l'erreur n'est pas assez petite, on calcule $A_{20} = Y_{n+1}^{(}m_2)$ puis A_{21} (polynôme de degré 1) et enfin A_{22} polynôme de degré 2 qui se calcule avec la formule de récurrence de Neville à partir de A_{11} et de A_{21}.

Les évaluations de la ligne j sont faites juste après le calcul de $A_{j0} = Y_{n+1}^{(}m_j)$. Chaque élément d'une ligne est calculé à partir de deux éléments situés une colonne en arrière, celui de la même ligne et celui de la ligne au dessus. La formule de récurrence de Neville conduit à la relation :

$$A_{j,i} = P_{j-i,\ldots,j} = A_{j,i-1} + \frac{A_{j,i-1} - A_{j-1,i-1}}{\left(\frac{h_{j-i}}{h_j}\right)^2 - 1} \text{ pour } i = 1, 1, \ldots, j \tag{19}$$

Avec la notation $P_{j-i,..,j}$, on voit que les points extrêmes sont $j-i$ et j, et les h^2 correspondant apparaissent dans la formule de récurrence.

La relation de récurrence peut être écrite avec les m_j :

$$A_{j,i} = A_{j,i-1} + \frac{A_{j,i-1} - A_{j-1,i-1}}{\left(\frac{m_j}{m_{j-i}}\right)^2 - 1} \text{ pour } i = 1, 1, \ldots, j \tag{20}$$

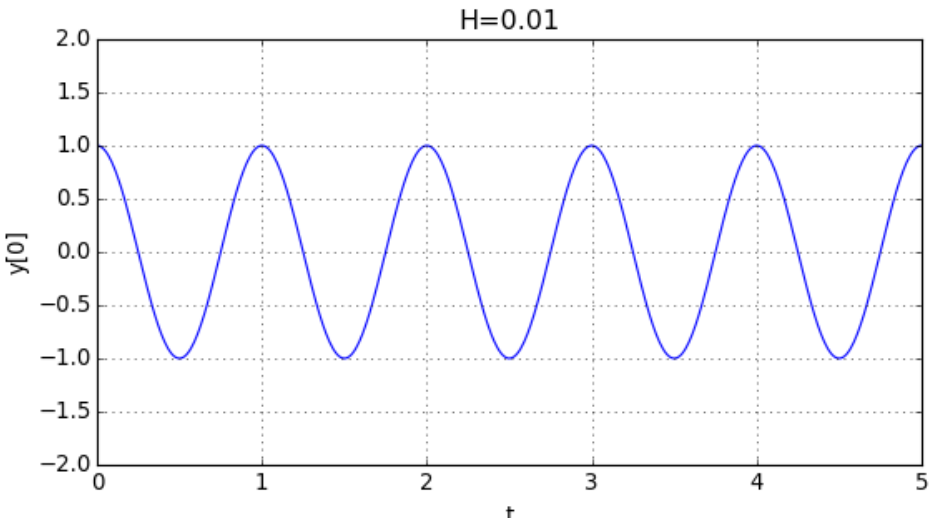
En pratique, on fixe une valeur maximale pour j. Il faudra donc prévoir un tableau fixe dont la taille est fixée par cette indice j maximal.

4.b. Implémentation à pas fixe

Avant de mettre en place un contrôle de la précision, on commence par faire une implémentation de test qui procède jusqu'à un indice fixe $j = jmax$. On remarque que le tableau A a en fait 3 dimensions car A_{ji} contient une liste de variables.

```
def bulirsch(systeme,jmax,Yi,T,H):
    A = numpy.zeros((jmax+1,jmax+1,len(Yi)))
    m = 2*(numpy.arange(jmax+1)+1)
```

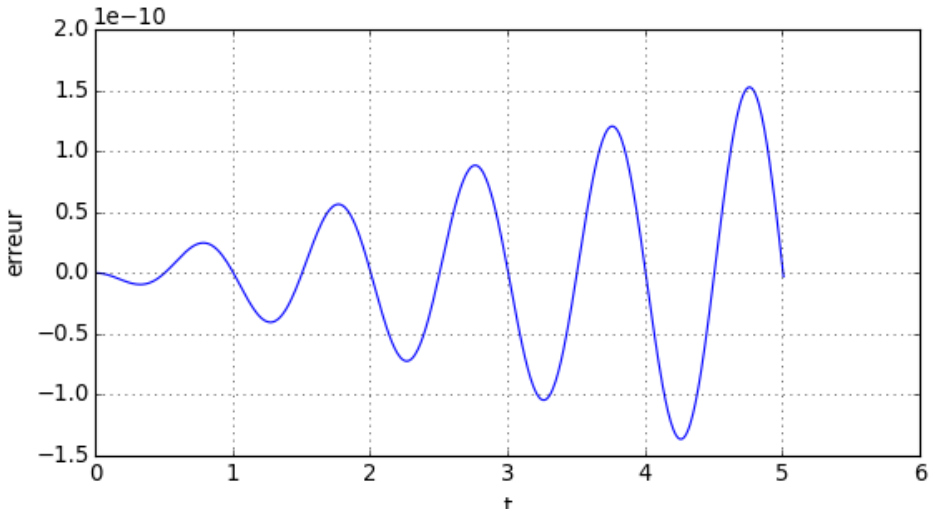
```
    Y = Yi
    t = 0.0
    liste_y = [Y]
    liste_t = [t]
    while t<T:
        for j in range(jmax+1):
            A[j][0] = pas_pointmilieu_modifie(systeme,H,t,Y,m[j])
            for i in range(1,j+1):
                correction = (A[j][i-1]-A[j-1][i-1])/((m[j]*1.0/m[j-i])**2-1)
                A[j][i] = A[j][i-1] + correction
        Y = numpy.array(A[jmax][jmax])
        t += H
        liste_y.append(Y)
        liste_t.append(t)
    return (numpy.array(liste_t),numpy.array(liste_y))

T = 5.0
H = 1.0e-2
Yi = [1.0,0]
jmax = 2
(t,tab_y) = bulirsch(oscillateur,jmax,Yi,T,H)
yt = tab_y.transpose()
x = yt[0]
figure(figsize=(8,4))
plot(t,x)
xlabel('t')
ylabel('y[0]')
axis([0,T,-2,2])
title('H=0.01')
grid()
```



La solution extacte étant connue, on peut calculer et tracer l'erreur globale (pour la première variable) :

```
erreur = x.copy()
for i in range(x.size):
    erreur[i] = x[i]-numpy.cos(2*numpy.pi*t[i])
figure(figsize=(8,4))
```

```
plot(t,erreur)
xlabel('t')
ylabel('erreur')
grid()
```



L'erreur locale est proportionnelle à la puissance suivante ([1]) :

$$\epsilon = aH^{2j_{max}+1} = aH^5 \tag{21}$$

On a donc ici l'équivalent d'une méthode d'ordre 4. Avec $j_{max} = 3$, on aurait une méthode d'ordre 6. Un avantage très important de la méthode de Bulirsch-Stoer est la possibilité d'augmenter l'ordre de la précision sans changer la méthode elle-même. Avec les méthodes classiques de Runge-Kutta et les méthodes à pas multiples cela est beaucoup plus difficile, car il faut au préalable déterminer les relations de récurrence pour les différents ordres que l'on souhaite utiliser.

4.c. Adaptation du pas

Afin de contrôler la précision de l'intégration, il est nécessaire de mettre en place une méthode d'adaptation des paramètres j_{max} et H en fonction de l'estimation de l'erreur. On commence par l'adaptation de H à j_{max} fixé, c'est-à-dire à ordre fixé.

Il faut tout d'abord établir un moyen de calculer une estimation de l'erreur. Pour cela, on doit tenir compte du fait que A_{ji} est en fait un tableau A_{jik} de N valeurs, où N est le nombre de variables. Pour calculer l'erreur, on introduit une tolérance absolue ϵ_a et une tolérance relative ϵ_r. Pour chaque variable d'indice k, on définit une échelle de tolérance par :

$$\epsilon_k = \epsilon_a + |A_{jjk}|\epsilon_r \tag{22}$$

On choisit le plus souvent une tolérance relative, mais on doit prévoir aussi une tolérance absolue qui entre en jeu lorsque la variable atteint des valeurs très faibles.

On peut alors définir une estimation de l'erreur réduite (sans dimensions) par :

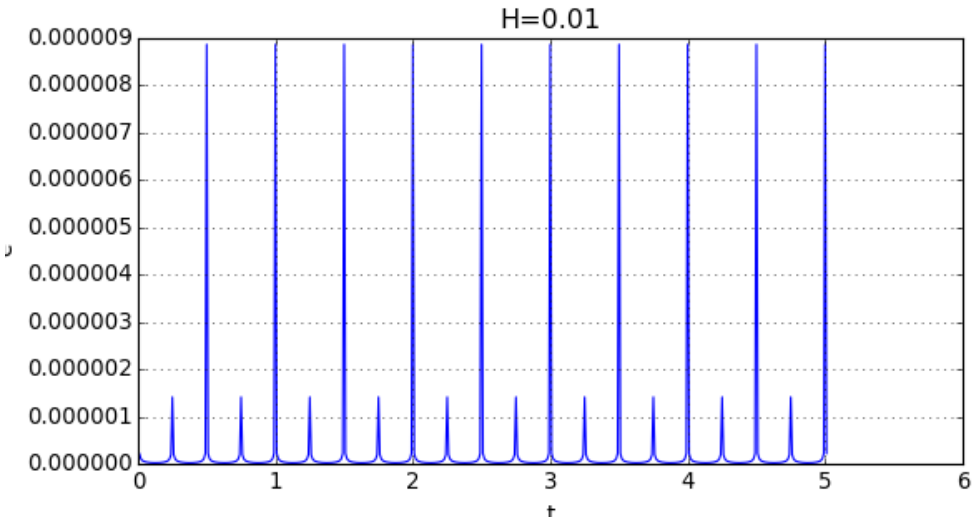
$$e_j = \sqrt{\frac{1}{N}\sum_{k=0}^{N-1}\left(\frac{|A_{jjk} - A_{jj-1k}|}{\epsilon_k}\right)^2} \tag{23}$$

On commence par une implémentation à pas fixe, avec un calcul de l'erreur lorsque $j = j_{max}$:

```
def bulirsch(systeme,jmax,Yi,T,H,atol=1e-6,rtol=1e-6):
    A = numpy.zeros((jmax+1,jmax+1,len(Yi)))
    m = 2*(numpy.arange(jmax+1)+1)
    Y = Yi
    t = 0.0
    liste_y = [Y]
    liste_t = [t]
    liste_e = [0]
    while t<T:
        for j in range(jmax+1):
            A[j][0] = pas_pointmilieu_modifie(systeme,H,t,Y,m[j])
            for i in range(1,j+1):
                correction = (A[j][i-1]-A[j-1][i-1])/((m[j]*1.0/m[j-i])**2-1)
                A[j][i] = A[j][i-1] + correction
        Y = numpy.array(A[jmax][jmax])
        e=0.0
        j = jmax
        N = len(Yi)
        for k in range(N):
            e += (abs(A[j][j][k]-A[j][j-1][k])/(atol+rtol*abs(A[j][j][k])))**2
        e = numpy.sqrt(e/N)
        t += H
        liste_y.append(Y)
        liste_t.append(t)
        liste_e.append(e)
    return (numpy.array(liste_t),numpy.array(liste_y),numpy.array(liste_e))
```

On choisit en général une tolérance relative. Cependant, lorsque les variables ont simultanément des valeurs très faibles (ce n'est pas le cas ici), la tolérance devient très faible et peut être impossible à satisfaire. On choisit donc une tolérance absolue de sécurité.

```
T = 5.0
H = 1.0e-2
Yi = [1.0,0]
jmax = 3
atol=1e-8
rtol=1e-6
(t,tab_y,e) = bulirsch(oscillateur,jmax,Yi,T,H,atol,rtol)
yt = tab_y.transpose()
x = yt[0]
figure(figsize=(8,4))
plot(t,e)
xlabel('t')
ylabel('e')
title('H=0.01')
grid()
```



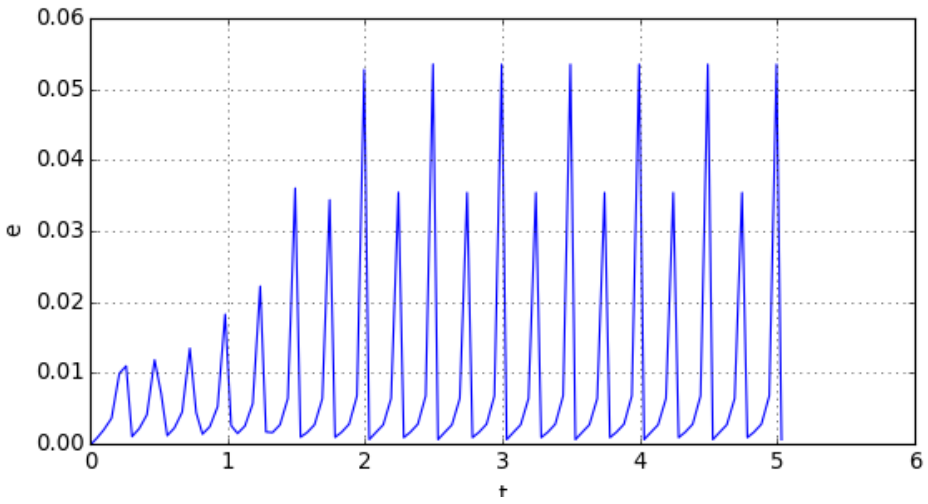
Il est important de remarquer qu'il s'agit d'une estimation de l'erreur locale, à ne pas confondre avec l'erreur globale (vraie) tracée plus haut. De plus, cette erreur est relative à la tolérance choisie. Ces valeurs d'erreur très inférieures à 1 montrent que, pour ce pas H, la tolérance choisie est très largement vérifiée. On a choisi $j_{max} = 3$, donc ce résultat est obtenu avec une extrapolation polynomiale de degré 3, à partir de 4 points. Dans ce cas, on peut augmenter le pas H afin de réduire la quantité de calculs. Inversement, si l'erreur est supérieure à 1 il faut réduire le pas. Si l'on reste à $j = j_{max}$ fixé, une méthode simple consiste à modifier H à chaque pas de temps en tenant compte de la loi d'échelle de l'erreur :

$$H \rightarrow H * S * \left(\frac{S}{e_j}\right)^{\frac{1}{2j+1}} \tag{24}$$

Le facteur S est choisi inférieur à 1. Voici un exemple :

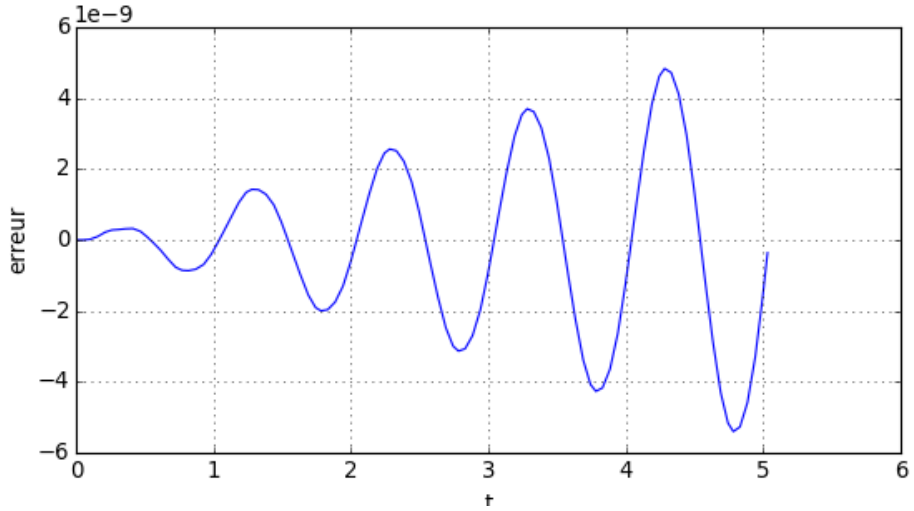
```
def bulirsch(systeme,jmax,Yi,T,H,atol=1e-6,rtol=1e-6,S=0.5):
    A = numpy.zeros((jmax+1,jmax+1,len(Yi)))
    m = 2*(numpy.arange(jmax+1)+1)
    Y = Yi
    t = 0.0
    liste_y = [Y]
    liste_t = [t]
    liste_e = [0]
    while t<T:
        for j in range(jmax+1):
            A[j][0] = pas_pointmilieu_modifie(systeme,H,t,Y,m[j])
            for i in range(1,j+1):
                correction = (A[j][i-1]-A[j-1][i-1])/((m[j]*1.0/m[j-i])**2-1)
                A[j][i] = A[j][i-1] + correction
        Y = numpy.array(A[jmax][jmax])
        e=0.0
        j = jmax
        N = len(Yi)
        for k in range(N):
            e += (abs(A[j][j][k]-A[j][j-1][k])/(atol+rtol*abs(A[j][j][k])))**2
        e = numpy.sqrt(e/N)
        t += H
        H = H*S*(S/e)**(1.0/(2*j+1))
```

```
        liste_y.append(Y)
        liste_t.append(t)
        liste_e.append(e)
    return (numpy.array(liste_t),numpy.array(liste_y),numpy.array(liste_e))

T = 5.0
H = 1.0e-2
Yi = [1.0,0]
jmax = 3
(t,tab_y,e) = bulirsch(oscillateur,jmax,Yi,T,H,atol,rtol)
yt = tab_y.transpose()
x = yt[0]
figure(figsize=(8,4))
plot(t,e)
xlabel('t')
ylabel('e')
grid()
```



Voici l'erreur globale vraie :

```
erreur = x.copy()
for i in range(x.size):
    erreur[i] = x[i]-numpy.cos(2*numpy.pi*t[i])
figure(figsize=(8,4))
plot(t,erreur)
xlabel('t')
ylabel('erreur')
grid()
```

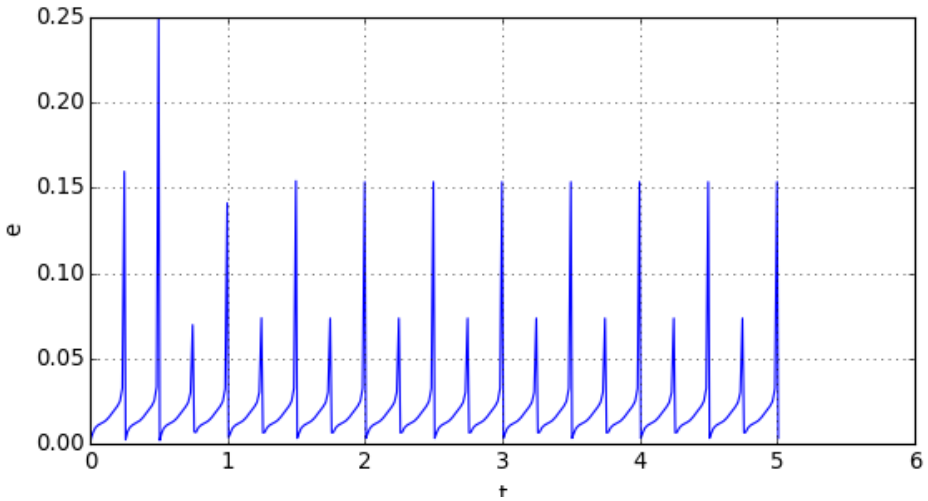


4.d. Adaptation de l'ordre

La méthode de Bulirsch-Stoer permet de modifier très facilement l'ordre. Il suffit de changer l'indice j_{max} auquel la subdivision du pas est stoppée. Une première approche consiste à stopper la subdivision lorsque l'erreur devient inférieure à 0.5. Le prix à payer dans l'implémentation de cette méthode est la nécessité de calculer l'erreur à chaque valeur de j. Le paramètre `jmax` désigne à présent la valeur maximale ne pouvant être dépassée.

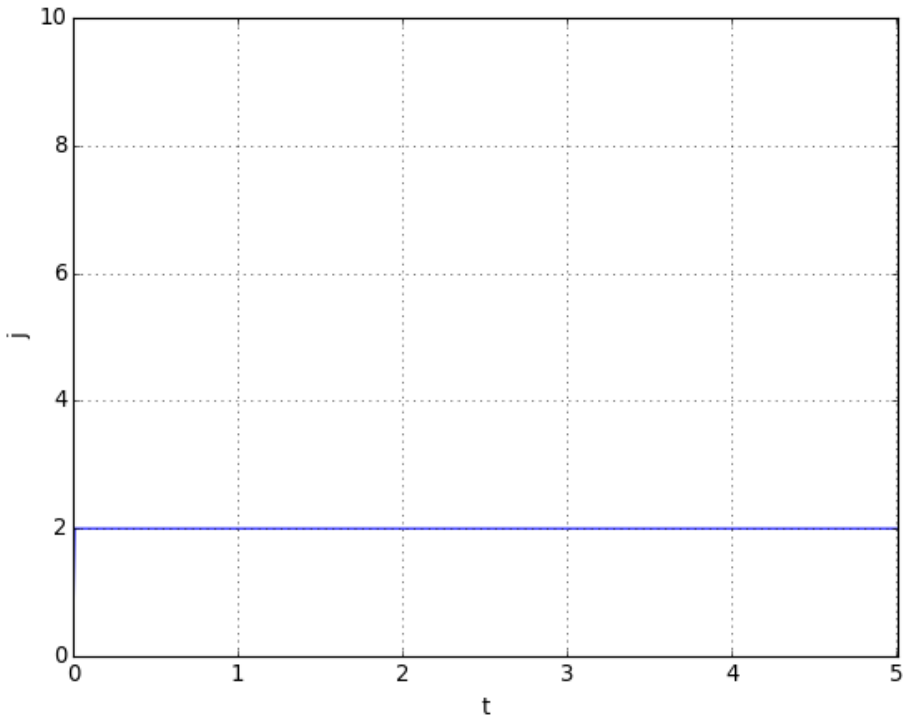
```
def bulirsch(systeme,jmax,Yi,T,H,atol=1e-6,rtol=1e-6,S=0.5):
    N = len(Yi)
    A = numpy.zeros((jmax+1,jmax+1,N))
    m = 2*(numpy.arange(jmax+1)+1)
    Y = Yi
    t = 0.0
    liste_y = [Y]
    liste_t = [t]
    liste_e = [0]
    liste_j = [0]
    while t<T:
        for j in range(jmax+1):
            A[j][0] = pas_pointmilieu_modifie(systeme,H,t,Y,m[j])
            for i in range(1,j+1):
                correction = (A[j][i-1]-A[j-1][i-1])/((m[j]*1.0/m[j-i])**2-1)
                A[j][i] = A[j][i-1] + correction
            e=0.0
            for k in range(N):
                e += (abs(A[j][j][k]-A[j][j-1][k])/(atol+rtol*abs(A[j][j][k])))**2
            e = numpy.sqrt(e/N)
            if e<0.5:
                break
        liste_j.append(j)
        t += H
        Y = numpy.array(A[j][j])
        H = H*S*(S/e)**(1.0/(2*j+1))
        liste_y.append(Y)
        liste_t.append(t)
        liste_e.append(e)
    return (numpy.array(liste_t),numpy.array(liste_y),numpy.array(liste_e),numpy.array(l
```

```
T = 5.0
H = 1.0e-2
Yi = [1.0,0]
jmax = 10
(t,tab_y,e,J) = bulirsch(oscillateur,jmax,Yi,T,H,atol,rtol)
yt = tab_y.transpose()
x = yt[0]
figure(figsize=(8,4))
plot(t,e)
xlabel('t')
ylabel('e')
grid()
```



Voici les valeurs maximales de j :

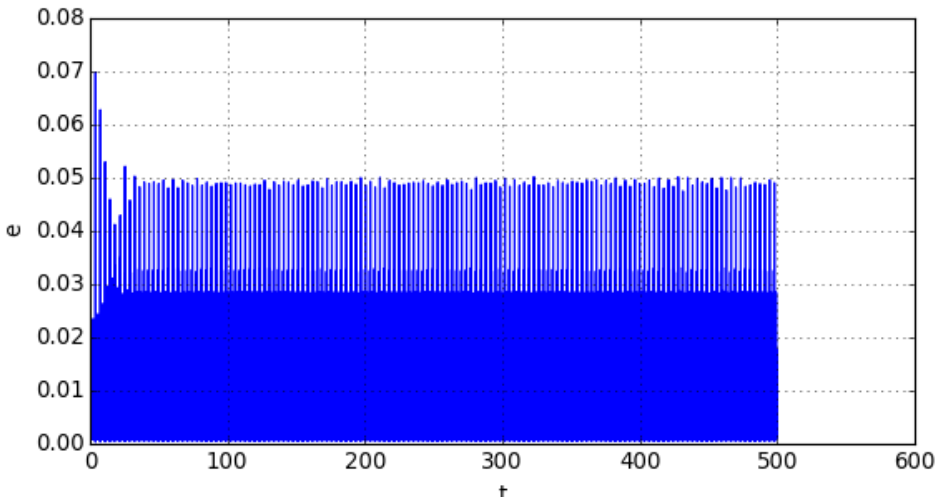
```
figure()
plot(t,J)
xlabel("t")
ylabel("j")
grid()
axis([0,t.max(),0,10])
```



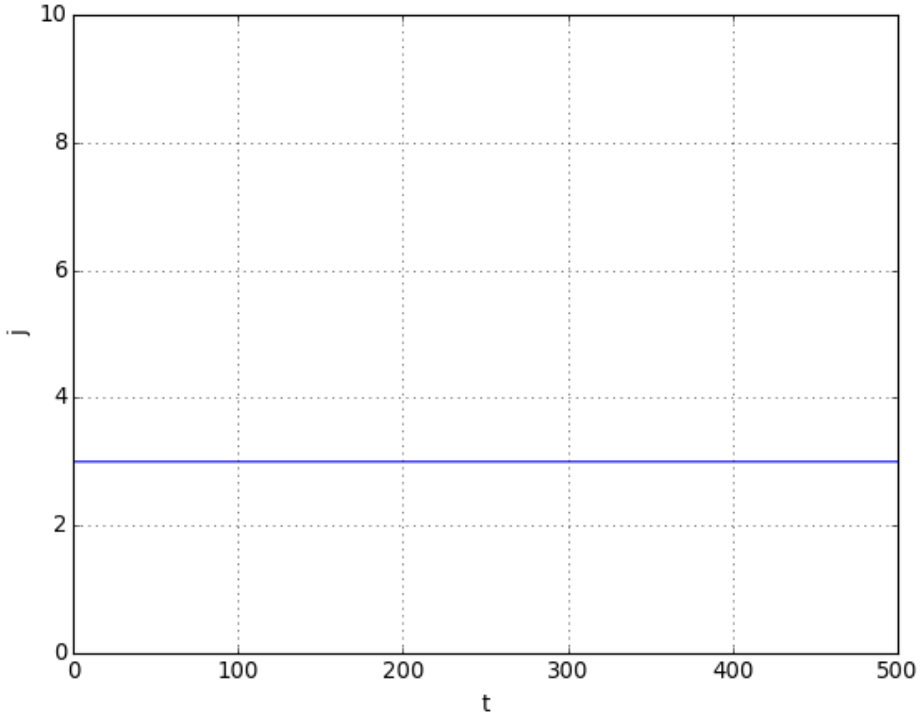
Pour cet exemple, un polynôme de degré 2 suffit pour assurer la tolérance demandée. Bien sûr, le degré du polynôme nécessaire dépend du pas H et de la tolérance choisie.

Voici un autre calcul avec une tolérance plus faible et une durée 100 fois plus grande :

```
T = 500.0
H = 1.0e-2
Yi = [1.0,0]
jmax = 10
atol=1e-12
rtol=1e-10
(t,tab_y,e,J) = bulirsch(oscillateur,jmax,Yi,T,H,atol,rtol)
yt = tab_y.transpose()
x = yt[0]
figure(figsize=(8,4))
plot(t,e)
xlabel('t')
ylabel('e')
grid()
```

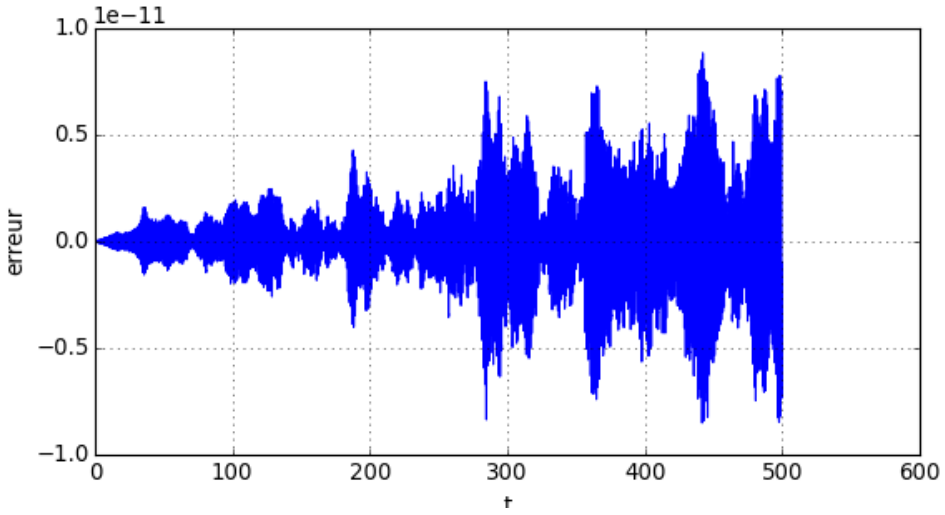



```
figure()
plot(t,J)
xlabel("t")
ylabel("j")
grid()
axis([0,t.max(),0,10])
```



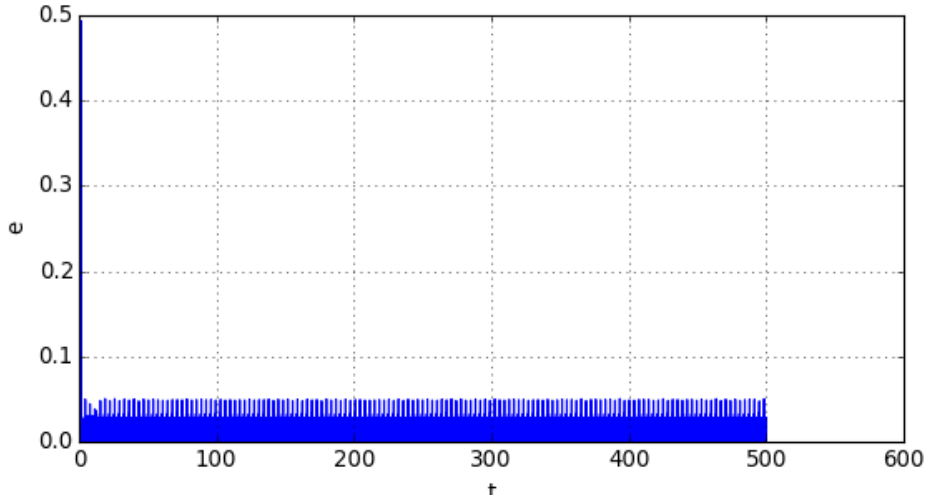

```
erreur = x.copy()
for i in range(x.size):
    erreur[i] = x[i]-numpy.cos(2*numpy.pi*t[i])
figure(figsize=(8,4))
```

```
plot(t,erreur)
xlabel('t')
ylabel('erreur')
grid()
```

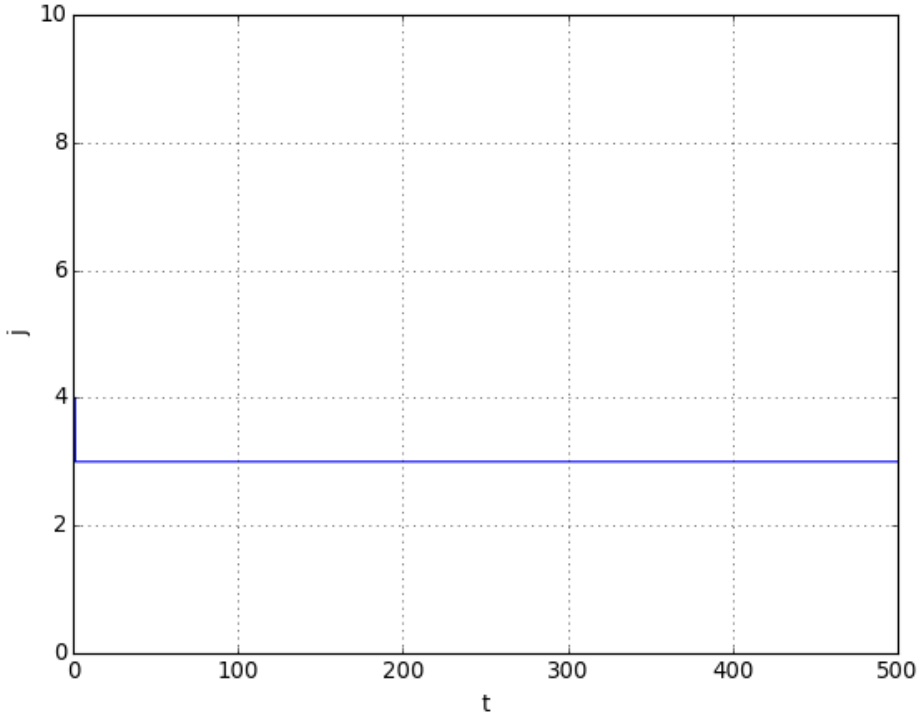


Pour la même tolérance, augmentons le pas H :

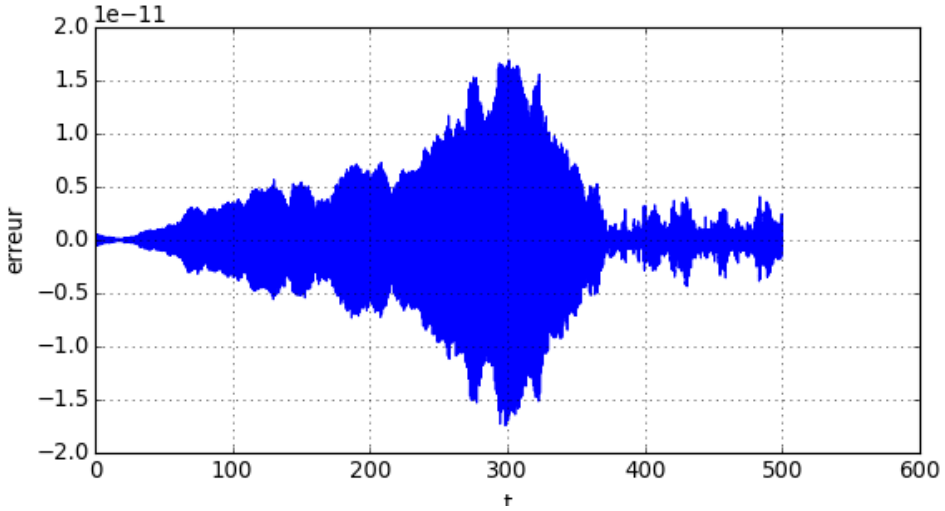
```
T = 500.0
H = 0.1
Yi = [1.0,0]
jmax = 10
atol=1e-12
rtol=1e-10
(t,tab_y,e,J) = bulirsch(oscillateur,jmax,Yi,T,H,atol,rtol)
yt = tab_y.transpose()
x = yt[0]
figure(figsize=(8,4))
plot(t,e)
xlabel('t')
ylabel('e')
grid()
```



```
figure()
plot(t,J)
xlabel("t")
ylabel("j")
grid()
axis([0,t.max(),0,10])
```




```
erreur = x.copy()
for i in range(x.size):
    erreur[i] = x[i]-numpy.cos(2*numpy.pi*t[i])
figure(figsize=(8,4))
plot(t,erreur)
xlabel('t')
ylabel('erreur')
grid()
```



Références

[1] W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, *Numerical recipes, the art of scientific computing*, (Cambridge University Press, 2007)