Introduction aux méthodes numériques d'étude des systèmes dynamiques

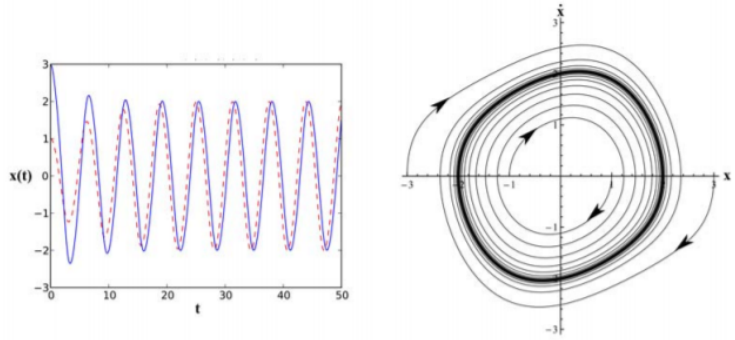

Figure 1 – Exemple de modélisation : oscillateur de Van Der Pol

1 Introduction

On considère un système dynamique dont l'équation différentielle est de la forme :

$$\frac{\mathrm{d}x}{\mathrm{d}t} = v(x)$$

Pour résoudre cette équation différentielle, et dans le cadre du programme, on utilise la méthode d'Euler.

On considère maintenant un système dynamique conservatif dont l'équation différentielle d'ordre 2 a la forme suivante :

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2} = a(x)$$

Pour résoudre numériquement cette équation différentielle, en utilisant la méthode d'Euler, on la transforme en deux équations différentielles d'ordre 1 :

$$\frac{dx}{dt} = v$$
$$\frac{dv}{dt} = a(x)$$

x est une variable de position, v la vitesse correspondante, et $a(x)$ l'accération en fonction de la position.

Un exemple d'équation de ce type est celle du pendule, avec θ pour la position (x), $\dot{\theta}$ pour la vitesse angulaire (v) :

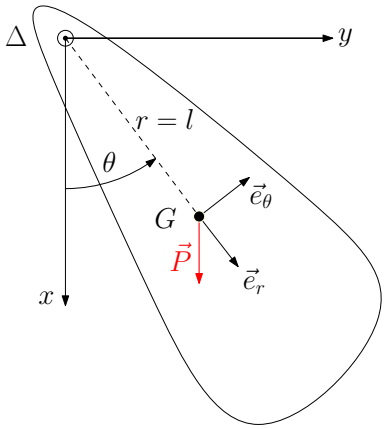

Figure 2 – Modélisation du pendule

$$\frac{dx}{dt} = v$$
$$\frac{dv}{dt} = -\omega_0^2 \sin(x)$$

On se propose d'étudier différentes méthodes d'Euler pour résoudre numériquement l'équation du pendule en prenant :

$$\omega_0 = 10\,\text{rad}\cdot\text{s}^{-1}$$

2 Méthode d'Euler explicite

2.1 Principe

Soit (x_0, v_0) une condition initiale pour $t = 0$. Les méthodes d'Euler consistent à calculer des valeurs approchées x_n, y_n de la solution pour cette condition initiale, aux instants suivants :

$$t_n = nh$$

La méthode d'Euler explicite est définie par :

$$x_{n+1} = x_n + hv_n$$
$$v_{n+1} = v_n + ha\,(x_n)$$

L'erreur réalisée avec cette méthode est représentée ci-dessous :

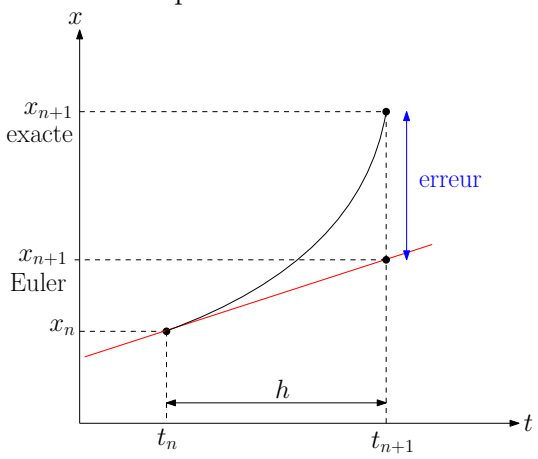

Figure 3 – Erreur commise avec Euler explicite

2.2 Application au pendule

La position correspond à θ, la vitesse correspond à la vitesse angulaire $\dot{\theta}$ et l'accélération vérfie :

$$a(x) = -\omega_0^2 \sin x$$

Le code donné figures (4) et (5) calcule N points par cette méthode pour **l'équation du pendule**, et renvoit 3 tableaux numpy :

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

w0=10
h=1e-3
N=100000
x0 = 1.5
v0 = 0

def euler(x0,v0,h,N):
    tab_x = np.zeros(N)
    tab_v = np.zeros(N)
    tab_t = np.arange(N)*h
    x = x0
    v = v0
    for i in range(N):
        tab_x[i] = x
        tab_v[i] = v
        (x,v) = (x+v*h,v-(w0**2)*np.sin(x)*h)
    return (tab_t,tab_x,tab_v)


(t,x,v) = euler(x0,v0,h,N)

plt.figure("réponse en fonction du temps")
plt.plot(t,x,label="position")
plt.plot(t,v,label="vitesse")
plt.xlabel("t")
plt.ylabel("position et vitesse")
plt.legend ()
plt.axis([0,10,-30,30])
plt.grid()
```

Figure 4 – Euler explicite pour le pendule - 1/2-

```python
plt.figure("Trajectoire de phase")
plt.plot(x,v/(2*np.pi),label="trajectoire de phase")
plt.xlabel("x")
plt.ylabel("v")
plt.axis([-3,3,-5,5])
plt.legend ()
plt.grid()

#comparaison avec odeint
def F(y,t):
   return (y[1],-(w0**2)*np.sin(y[0]))
theta0=x0
theta_prime_0=v0
y_ini = (theta0,theta_prime_0)
y_sol = odeint (F,y_ini ,t)
theta = y_sol [: ,0]
theta_prime = y_sol [: ,1]

plt.figure("aymetrique vs odeint")
plt.plot(t,x,"b-", label="position avec Euler asymétrique")
plt.plot (t,theta , "r--",label =" position avec odeint ")
plt.xlabel ("t (s)")
plt.ylabel (" theta ( degrés)")
plt.axis([0,10,-3,3])
plt.legend ()
plt.grid ()
```

Figure 5 – Euler explicite pour le pendule - 2/2-

Le graphe obtenu est le suivant :

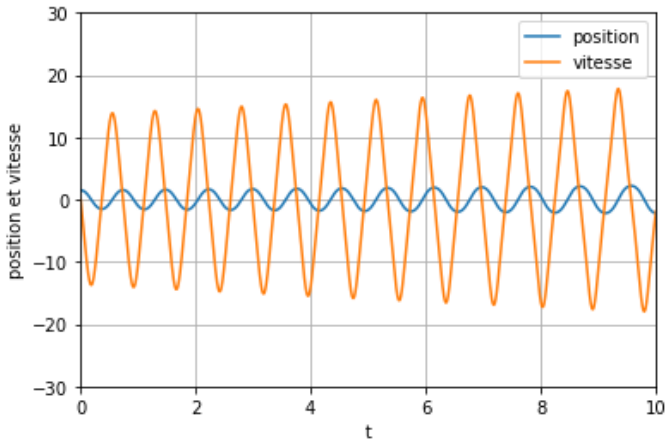

Figure 6 – Divergence de la position et la vitesse avec la méthode d'Euler explicite

Et le portrait de phase :

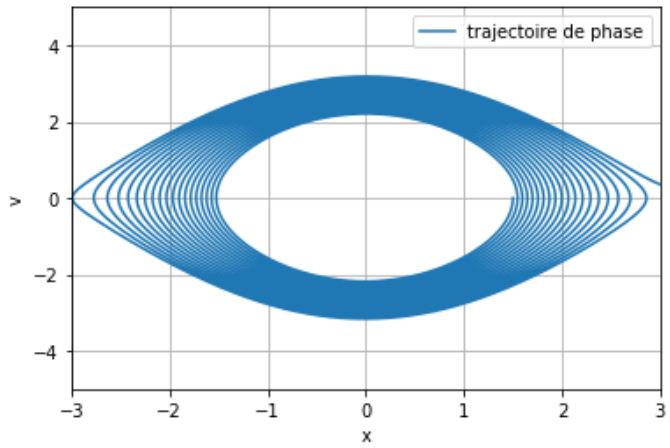

Figure 7 – Portrait de phase avec Euler explicite

La comparaison avec le résultat obtenu avec odeint donne :

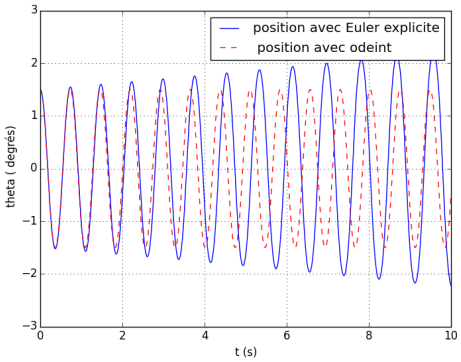

Figure 8 – Odeint vs Euler explicite

L'amplitude des oscillations augmente au cours du temps (le mouvement n'est pas périodique). Cela met en évidence l'instabilité de la méthode d'Euler explicite pour ce type de problème. On démontre en effet que cette méthode est instable pour l'oscillateur harmonique, en considérant les valeurs propres se son propagateur P, qui est la matrice permettant de calculer :

$$\begin{pmatrix} x_{n+1} \\ v_{n+1} \end{pmatrix} = P \begin{pmatrix} x_n \\ v_n \end{pmatrix}$$

Cette matrice n'est définie que si le système est linéaire. Pour un système non linéaire, on l'obtient en linéarisant au voisinage d'une position d'équilibre. Pour l'oscillateur harmonique, qui correspond au voisinage de la position $x = 0$ de l'équation du pendule, le propagateur est :

$$P = \begin{pmatrix} 1 & h \\ -\omega^2 h & 1 \end{pmatrix}$$

La méthode est stable si et seulement si toutes les valeurs propres du propagateur (en général complexes) ont un module inférieur ou égal à 1.

Dans le cas présent, les deux valeurs propres ont un module strictement supérieur à 1 [1], ce qui démontre l'instabilité de la méthode d'Euler explicite.

3 Méthode d'Euler asymétrique

3.1 Principe

Pour les système dynamiques conservatifs, il existe une modification de la méthode d'Euler qui permet de la rendre stable :

$$\begin{aligned} x_{n+1} &= x_n + h v_n \\ v_{n+1} &= v_n + h a\left(x_{n+1}\right) \end{aligned}$$

L'erreur liée à cette méthode est donnée ci-dessous :

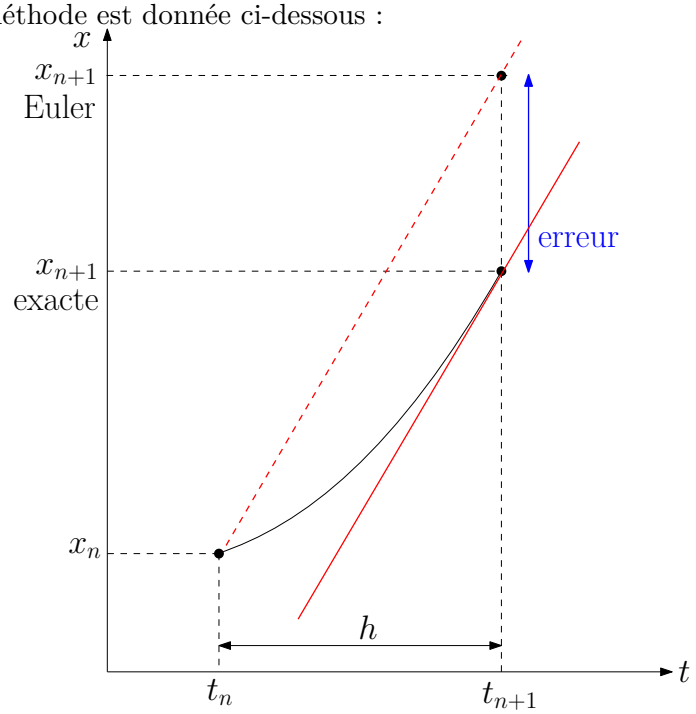

Figure 9 – Erreur commise avec Euler explicite

La différence avec la méthode explicite est l'évaluation de l'« accélération », qui se fait avec la position x_{n+1} et non pas x_n. Cette méthode est dite asymétrique en raison du traitement différent des variables positions et vitesses.

1. Les valeurs propres vérifient : $(1-\lambda)^2 = -\omega^2 h^2 \Rightarrow |\lambda| = \sqrt{1+(\omega h)^2} > 1$

3.2 Application au pendule

Voici une fonction réalisant cette méthode pour l'équation du pendule, et le résultat du calcul pour les mêmes paramètres que plus haut :

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

w0=10
h=0.01
N=100000
x0 = 1.5
v0 = 0

def euler_asym(x0,v0,h,N):
    tab_x = np.zeros(N)
    tab_v = np.zeros(N)
    tab_t = np.arange(N)*h
    x = x0
    v = v0
    for i in range(N):
        tab_x[i] = x
        tab_v[i] = v
        x= x+v*h
        v = v-(w0**2)*np.sin(x)*h
    return (tab_t,tab_x,tab_v)

w0=10
h=0.01
N=100000
x0 = 1.5
v0 = 0
(t,x,v) = euler_asym(x0,v0,h,N)

plt.figure("réponse en fonction du temps")
plt.plot(t,x,label="position")
plt.plot(t,v,label="vitesse")
plt.xlabel("t")
plt.ylabel("position et vitesse")
plt.legend ()
plt.axis([0,10,-30,30])
plt.grid()
```

Figure 10 – Code Euler asymetrique

```python
#comparaison avec odeint
def F(y,t):
   return (y[1],-(w0**2)*np.sin(y[0]))
theta0=1.5
theta_prime_0=0
y_ini = (theta0,theta_prime_0)
y_sol = odeint (F,y_ini ,t)
theta = y_sol [: ,0]
theta_prime = y_sol [: ,1]

plt.figure("aymetrique vs odeint")
plt.plot(t,x,"b-", label="position avec Euler asymétrique")
plt.plot (t,theta , "r--",label =" position avec odeint ")
plt.xlabel ("t (s)")
plt.ylabel (" theta ( degrés)")
plt.axis([0,10,-2,2])
plt.legend ()
plt.grid ()

```

Figure 11 – Code Euler asymetrique

Le graphe obtenu est le suivant :

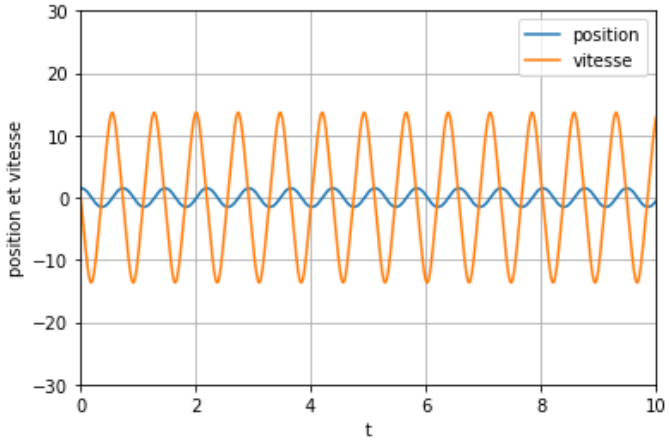

Figure 12 – Psosition et vitesse pour Euler asymétrique

Et le portrait de phase :

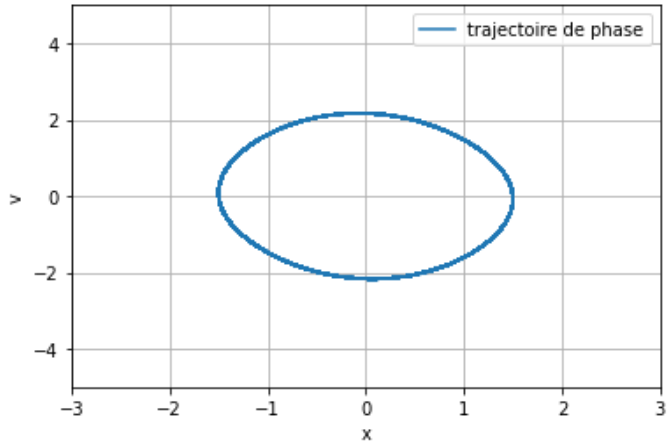

Figure 13 – Portrait de phase Euler asymetrique

La comparaison avec le résultat obtenu avec odeint donne :

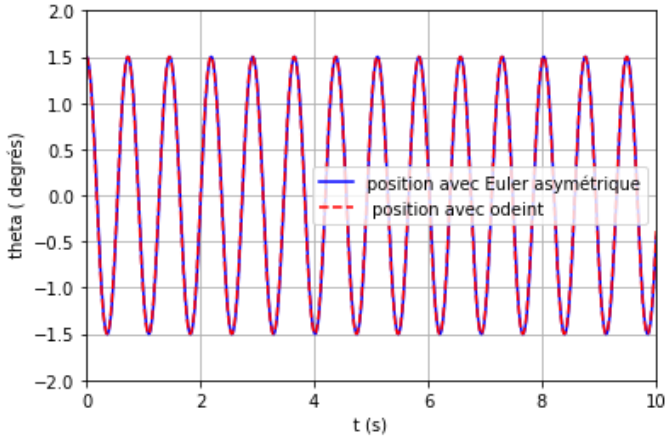

Figure 14 – Odeint vs Euler asymétrique

On obtient bien en apparence une solution périodique, avec conservation de l'énergie. Le propagateur pour l'oscillateur harmonique est :

$$P = \begin{pmatrix} 1 & h \\ -\omega^2 h & 1-\omega^2 h^2 \end{pmatrix}$$

La stabilité est vérifiée si $h\omega < 2$ [2], soit dans notre cas $h < 1/5$. Si cette condition est vérifiée, les valeurs propres ont même un module exactement égal à 1, ce qui explique la périodicité de la solution obtenue. C'est la méthode idéale pour les systèmes conservatifs.

4 Méthode d'Euler implicite

4.1 Principe

La méthode d'Euler implicite est définie par :

$$\begin{aligned} x_{n+1} &= x_n + hv_{n+1} \\ v_{n+1} &= v_n + ha\left(x_{n+1}\right) \end{aligned}$$

Le calcul de x_{n+1}, v_{n+1} nécessite de résoudre l'équation :

$$x_{n+1} + h^2 a\left(x_{n+1}\right) = x_n + hv_n$$

Lorsque le système est non linéaire, comme c'est le cas pour l'équation du pendule, il faut donc résoudre une équation non linéaire à chaque pas de temps. On utilise pour cela la méthode de Newton.

Lorsque x_{n+1} est obtenu, on calcule :

$$v_{n+1} = \frac{x_{n+1} - x_n}{h}$$

4.2 Méthode de Newton

Soit f une fonction continue d'une variable réelle u à valeurs réelles, de classe C^1 sur un intervalle $[a, b]$. On suppose qu'il existe une valeur u_r et une seule sur cet intervalle telle que $f\left(u_r\right) = 0$. La méthode de Newton permet d'obtenir une approximation numérique de u_r. On part d'une valeur u_0 dans l'intervalle $[a, b]$ et on considère la tangente à la courbe en ce point. L'intersection de cette tangente avec l'axe Ox donne u_1.

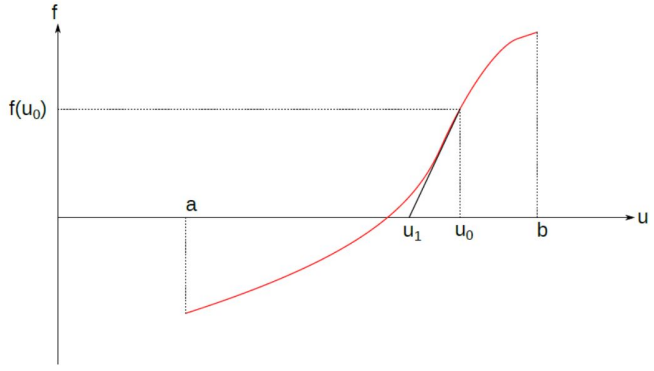

Figure 15

L'opération est répétée, ce qui donne les abcisses u_2, u_3, etc. La suite est définie par la relation de récurrence suivante :

$$u_{n+1} = u_n - \frac{f\left(u_n\right)}{f'\left(u_n\right)}$$

2. En effet, Les valeurs propres sont solutions de l'équation : $\lambda^2 + (h^2\omega^2 - 2)\lambda + 1 = 0$. Si $h\omega < 2$ on obtient deux racines complexes conjuguées dont le module est égal à 1. Dans ce cas le schéma est stable. Si $h\omega > 2$ on obtient deux racines réelles dont le module est strictement supérieur à 1 et donc le schéma est instable

Cette suite converge vers la racine u_r. La méthode de Newton converge beaucoup plus rapidement que la méthode de dichotomie (mais elle peut dans certains cas échouer). En pratique, on stoppe l'itération lorsque

$$|f(u_p)| < \epsilon$$

où ϵ est une tolérance que l'on choisit en fonction de la précision souhaitée. Pour utiliser la méthode de Newton, il faut connaître la fonction f et sa dérivée f'. Voici une fonction qui implémente la méthode de Newton. On prévoit un nombre maximal d'itérations.

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

def newton(f,df,u0,epsilon):
    u = u0
    y = f(u)
    iter = 0
    while abs(y) > epsilon and iter < 100:
        u = u-y/df(u)
        y = f(u)
        iter += 1
    if iter==100:
        raise Exception("maximum d'iteration atteint")
    print("%d iterations"%iter)
    return u

def f(u):
    return u*u-2
def df(u):
    return 2*u

r = newton(f,df,1.0,1e-15)
print(r)
print(math.sqrt(2))
```

Figure 16 – Exemple de méthode Newton : recherche des racines de $u^2 - 2$

On trouve un nombre d'itération égal à 5, avec $r \approx \sqrt{2} = 1.4142135623730951$.

Ainsi, la convergence vers la solution approchée à la précision du type flottant se fait en seulement 5 itérations. Bien sûr, le nombre d'itérations dépend du point initial. Pour cet exemple, la méthode de Newton échoue si la valeur initiale est $u_0 = 0$, car la tangente en ce point ne coupe pas l'axe Ox.

4.3 Application au pendule

Pour la méthode d'Euler implicite, la fonction dont on cherche la racine est :

$$f(u) = u - h^2 a(u) - (x_n + h v_n)$$

Pour l'équation du pendule, la fonction f et sa dérivée sont :

$$f(u) = u + \omega_0^2 h^2 \sin(u) - (x_n + h v_n)$$
$$f'(u) = 1 + \omega_0^2 h^2 \cos(u)$$

Voici une fonction qui fait le calcul. Elle renvoit aussi le nombre moyen d'itérations dans la méthode de Newton.

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

w0=10
h=0.001
N=100000
x0 = 1.5
v0 = 0

def euler_implicite(x0,v0,h,N,epsilon):
    tab_x = np.zeros(N)
    tab_v = np.zeros(N)
    tab_t = np.arange(N)*h
    x = x0
    v = v0

    b = w0**2*h**2
    n_iter = 0
    for i in range(N):
        tab_x[i] = x
        tab_v[i] = v
        u = x
        d = x+h*v
        f = u+b*np.sin(u)-d
        while abs(f) > epsilon:
            u = u-f/(1+b*np.cos(u))
            f = u+b*np.sin(u)-d
            n_iter += 1
        v = (u-x)/h
        x = u
    return (tab_t,tab_x,tab_v,n_iter*1.0/N)



(t,x,v,n_iter) = euler_implicite(x0,v0,h,N,1e-8)
```

Figure 17 – Code Euler asymetrique

```python
plt.figure("réponse en fonction du temps")
plt.plot(t,x,label="position")
plt.plot(t,v,label="vitesse")
plt.xlabel("t")
plt.ylabel("position et vitesse")
plt.legend ()
plt.axis([0,10,-30,30])
plt.grid()

plt.figure("Trajectoire de phase")
plt.plot(x,v/(2*np.pi),label="trajectoire de phase")
plt.xlabel("x")
plt.ylabel("v")
plt.axis([-1.7,1.7,-2.5,2.5])
plt.legend ()
plt.grid()

print(n_iter)

#comparaison avec odeint

def F(y,t):
   return (y[1],-(w0**2)*np.sin(y[0]))
theta0=x0
theta_prime_0=v0
y_ini = (theta0,theta_prime_0)
y_sol = odeint (F,y_ini ,t)
theta = y_sol [: ,0]
theta_prime = y_sol [: ,1]

plt.figure("euler implicite vs odeint")
plt.plot(t,x,"b-", label="position avec Euler implicite")
plt.plot (t,theta , "r--",label =" position avec odeint ")
plt.xlabel ("t (s)")
plt.ylabel (" theta (degrés)")
plt.axis([0,10,-3,3])
plt.legend ()
plt.grid ()
```

Figure 18 – Code Euler asymetrique

Le graphe obtenu est le suivant :

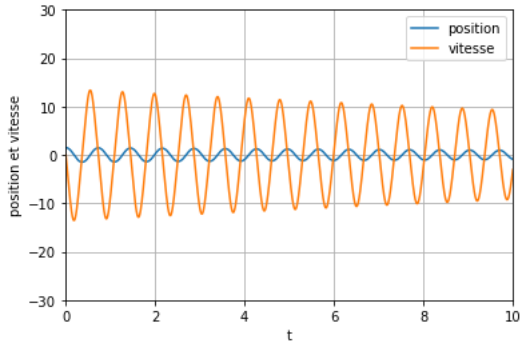

Figure 19 – Décroissance de la position et la vitesse avec la méthode d'Euler implicite

Et le portrait de phase :

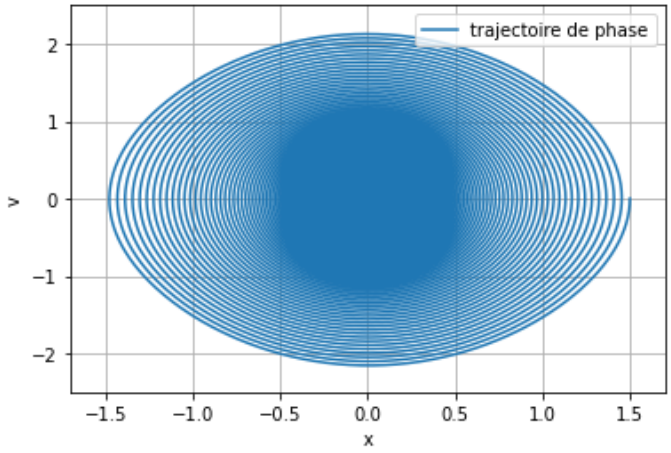

Figure 20 – Portrait de phase Euler implicite

La comparaison avec le résultat obtenu avec odeint donne :

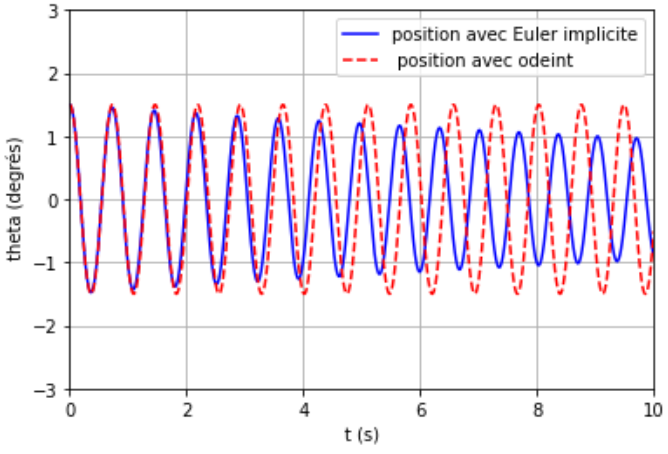

Figure 21 – Odeint vs Euler asymétrique

Le nombre d'itération est : $r = 1$

On voit que la méthode de Newton converge en une seule itération, car la valeur de départ est très proche de la racine.

La solution approchée n'est pas périodique : on observe une réduction de l'amplitude des oscillations, c'est-à-dire une diminution de l'énergie. Pour ce type de problème conservatif, la méthode implicite est donc tout aussi mauvaise que la méthode explicite.

Le propagateur pour l'oscillateur harmonique est :

$$P = \frac{1}{1+\omega^2 h^2}\begin{pmatrix} 1 & h \\ -\omega^2 h & 1-\omega^2 h^2 \end{pmatrix}$$

Ses deux valeurs propres ont un module strictement inférieur à 1. La méthode d'Euler implicite est donc stable (quelque soit le pas de temps). Cependant, elle n'est pas adapté au problème du pendule car la solution obtenue présente une décroissance de l'énergie. La méthode d'Euler asymétrique considérée plus haut n'a pas cet inconvénient, car les valeurs propres ont un module exactement égal à 1.

5 Méthode Runge-Kutta (HP)

5.1 Principe de la méthode

Cette méthode plus complexe fait intervenir quatre fois plus de calculs que la méthode d'Euler (pour une équation différentielle du premier ordre), mais est beaucoup plus fiable.

Voici les étapes de calculs, pour une équation différentielle d'ordre 1 :

$$\begin{cases} y(t=0) = y_0 \\ k_{10} = f\left(t_0, y_0\right) \\ k_{20} = f\left(t_0 + \frac{h}{2}, y_0 + h\frac{k_1}{2}\right) \\ k_{30} = f\left(t_0 + \frac{h}{2}, y_0 + h\frac{k_2}{2}\right) \\ k_{40} = f\left(t_0 + h, y_0 + hk_3\right) \\ y_1 = y_0 + \frac{h}{6}\left(k_{10} + 2k_{20} + 2k_{30} + k_{40}\right) \\ \ldots \end{cases}$$

II y a donc quatre coefficients à calculer (méthode d'ordre 4, il existe également la méthode RK2) qui correspondent à l'évaluation de 4 pentes à des instants différents entre t et $t + \mathrm{d}t$ (une évaluation en t, deux évaluations en $t + \frac{\mathrm{d}t}{2}$, une évaluation en $t + \mathrm{d}t$). Voici un graphique qui montre le lien entre les différents coefficients k :

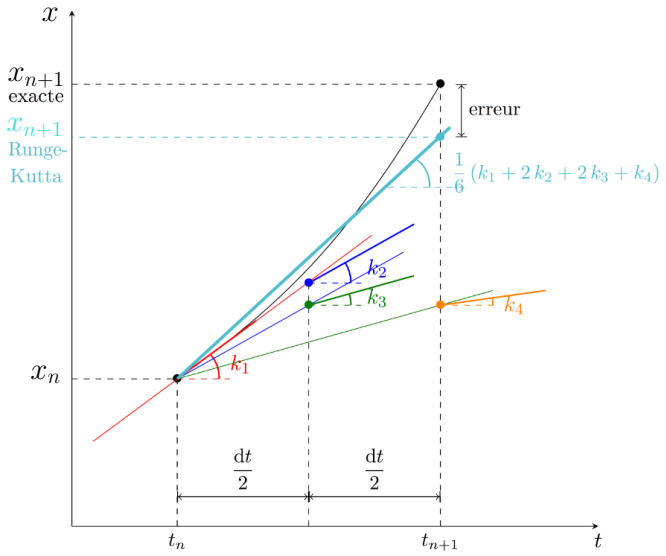

Figure 22 – Erreur Runge-Kutta

5.2 RK4 pour le pendule

Dans le cas du pendule, on a une équation différentielle d'ordre 2, que l'on transforme en 2 équations différentielles d'ordre 2, on doit donc introduire 8 coefficients RK4 :

$$\frac{dx}{dt} = v = \text{diffx}(t,x,v)$$
$$\frac{dv}{dt} = -\omega_0^2 \sin(x) = \text{diffv}(t,x,v)$$

$$\begin{cases} x(t=0) = x_0 \\ v(t=0) = v_0 \\ k_{1x} = \text{diffx}\left(t_0, x_0, v_0\right) \times h \\ k_{1v} = \text{diffv}\left(t_0, x_0, v_0\right) \times h \\ k_{2x} = \text{diffx}\left(t_0 + \frac{h}{2}, x_0 + h\frac{k_{1x}}{2}, v_0 + h\frac{k_{1v}}{2}\right) \\ k_{2v} = \text{diffv}\left(t_0 + \frac{h}{2}, x_0 + h\frac{k_{1x}}{2}, v_0 + h\frac{k_{1v}}{2}\right) \\ k_{3x} = \text{diffx}\left(t_0 + \frac{h}{2}, x_0 + h\frac{k_{2x}}{2}, v_0 + h\frac{k_{2v}}{2}\right) \\ k_{3v} = \text{diffv}\left(t_0 + \frac{h}{2}, x_0 + h\frac{k_{2x}}{2}, v_0 + h\frac{k_{2v}}{2}\right) \\ k_{4x} = \text{diffx}\left(t_0 + h, x_0 + hk_{3x}, v_0 + hk_{3v}\right) \\ k_{4v} = \text{diffv}\left(t_0 + h, x_0 + hk_{3x}, v_0 + hk_{3v}\right) \\ x_1 = x_0 + \frac{h}{6}\left(k_{1x} + 2k_{2x} + 2k_{3x} + k_{4x}\right) y_1 = y_0 + \frac{h}{6}\left(k_{1v} + 2k_{2v} + 2k_{3v} + k_{4v}\right) \\ \ldots \end{cases}$$

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

w0=10
h=1e-3
N=10000
x0 = 1.5
v0 = 0

def diffx(t,x,v):
    return v

def diffv(t,x,v):
    return -(w0**2)*np.sin(x)

def rk4(h,ti,xi,vi):
    k1x = diffx(ti,xi,vi);
    k1v = diffv(ti,xi,vi);
    k2x = diffx(ti+h/2,xi+h*k1x/2,vi+h*k1v/2);
    k2v = diffv(ti+h/2,xi+h*k1x/2,vi+h*k1v/2);
    k3x = diffx(ti+h/2,xi+h*k2x/2,vi+h*k2v/2);
    k3v = diffv(ti+h/2,xi+h*k2x/2,vi+h*k2v/2);
    k4x = diffx(ti+h,xi+h*k3x,vi+h*k3v);
    k4v = diffv(ti+h,xi+h*k3x,vi+h*k3v);
    xf = xi + (h/6)*(k1x + 2*k2x + 2*k3x + k4x);
    vf = vi + (h/6)*(k1v + 2*k2v + 2*k3v + k4v);
    tf = ti+h;
    return   xf,vf,tf


t = np.zeros(N)
x = np.zeros(N)
v = np.zeros(N)
```

Figure 23 – Code Runge-Kutta pour le pendule

```
t[0] = 0
x[0] = x0
v[0] = v0
for i in range (1,N):
    x[i],v[i],t[i] = rk4(h,t[i-1],x[i-1],v[i-1]);

plt.figure("angle et vitesse")
plt.plot(t,x,label="angle")
plt.plot(t,v,label="vitesse")
plt.title("mouvement d'un pendule simple")
plt.xlabel("temps")
plt.ylabel("angle")
plt.legend ()
plt.axis([0,10,-20,20])
plt.grid()

plt.figure("Trajectoire de phase")
plt.plot(x,v/(2*np.pi),label="trajectoire de phase")
plt.xlabel("x")
plt.ylabel("v")
plt.axis([-3,3,-5,5])
plt.legend ()
plt.grid()

#comparaison avec odeint
def F(y,t):
   return (y[1],-(w0**2)*np.sin(y[0]))
theta0=x0
theta_prime_0=v0
y_ini = (theta0,theta_prime_0)
y_sol = odeint (F,y_ini ,t)
theta = y_sol [: ,0]
theta_prime = y_sol [: ,1]
```

Figure 24 – Code Runge-Kutta pour le pendule

```
70  plt.figure("aymetrique vs odeint")
71  plt.plot(t,x,"b-", label="position avec Euler asymétrique")
72  plt.plot (t,theta , "r--",label =" position avec odeint ")
73  plt.xlabel ("t (s)")
74  plt.ylabel (" theta ( degrés)")
75  plt.axis([0,10,-3,3])
76  plt.legend ()
77  plt.grid ()
```

Figure 25 – Code Runge-Kutta pour le pendule

Le graphe obtenu est le suivant :

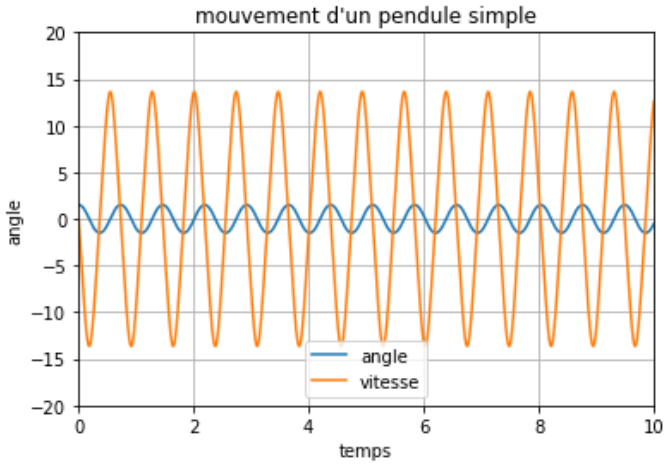

Figure 26 – Angle et vitesse avec la méthode RK4

Et le portrait de phase :

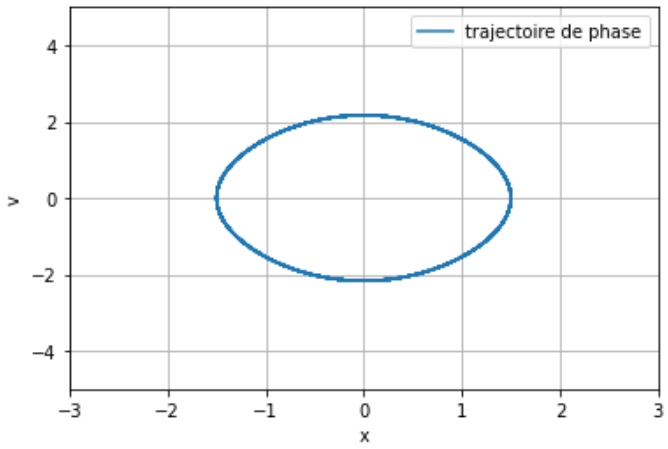

Figure 27 – Portrait de phase RK4

La comparaison avec le résultat obtenu avec odeint donne :

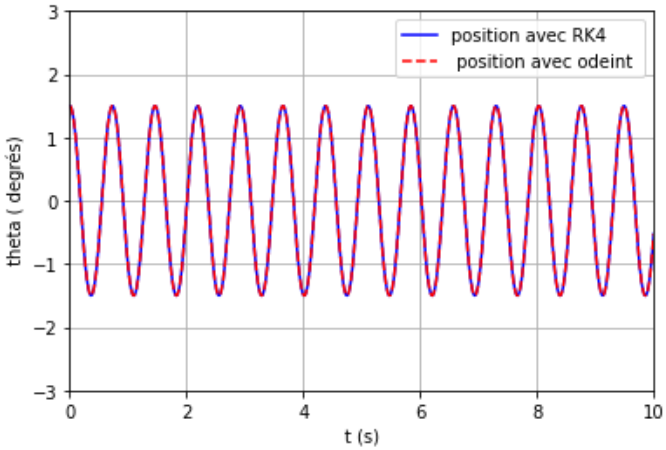

Figure 28 – Odeint vs RK4