TP 1 : Méthodes mathématiques et numériques

Boris Alexandre Baudel - Université du Maine - Master de Physique

October 2023

1 Introduction

Dans ce travail pratique, nous explorons diverses méthodes mathématiques et numériques axées sur la résolution des équations différentielles, un domaine crucial en physique et dans de nombreuses autres disciplines scientifiques. Nous débutons par une analyse approfondie des équations différentielles d'ordre supérieur, en les transformant en systèmes d'équations du premier ordre, facilitant ainsi leur résolution numérique. Des méthodes spécifiques, telles que la méthode d'Euler et les méthodes de Runge-Kutta d'ordre 2 et 4, sont ensuite examinées et appliquées pour résoudre ces équations de manière approchée.

En outre, nous mettons en œuvre des programmes en Python pour illustrer ces méthodes, en comparant les solutions numériques obtenues avec les solutions analytiques exactes lorsque celles-ci sont disponibles. Cette approche pratique permet de mieux comprendre l'impact du choix de la méthode et du nombre de points sur la précision de la solution numérique.

2 Résolution numérique des équations différentielles

2.1 Équation différentielle d'ordre n

Considérons une équation différentielle d'ordre $n > 1$ de la forme suivante :

$$x^{(n)}(t) \equiv \frac{d^{n-1}x}{dt^{n-1}} = \varphi\left(t, x(t), x^{(1)}(t), \dots, x^{(n-1)}(t)\right), \quad \forall t \in [a, b], \tag{1}$$

où $a, b \in \mathbb{R}$. Introduisons de nouvelles fonctions $y_i(t)$ avec $i \in \{1, 2, \dots, n\}$ telles que :

$$\begin{aligned} y_1(t) &\equiv x(t), \\ y_2(t) &\equiv x^{(1)}(t), \\ &\vdots \\ y_n(t) &\equiv x^{(n-1)}(t). \end{aligned} \tag{2}$$

On peut alors écrire l'équation différentielle (1) comme un système de n équations différentielles d'ordre 1 :

$$\begin{cases} \frac{dy_1}{dt} = y_2(t) \\ \vdots \\ \frac{dy_{n-1}}{dt} = y_n(t) \\ \frac{dy_n}{dt} = \varphi(t, y_1(t), y_2(t), \dots, y_n(t)) \end{cases} \tag{3}$$

qu'on peut encore mettre sous la forme :

$$\frac{dy}{dt} = f(t, y(t)), \quad \forall t \in [a, b], \tag{4}$$

avec

$$\begin{cases} y_i(t) = x^{(i-1)}(t), & \forall i \in \{1, \dots, n\} \\ f_i(t, y(t)) = y_{i+1}, & \forall i \in \{1, \dots, n-1\} \\ f_n(t, y(t)) = \varphi(t, y_1(t), y_2(t), \dots, y_n(t)) \end{cases} \tag{5}$$

Il faut donc résoudre l'équation (4) avec les conditions initiales $(y_1(a), \dots, y_n(a))$, ce qui revient à dire, compte tenu de l'équation (2), qu'il faut connaître $x(a), x^{(1)}(a), \dots, x^{(n-1)}(a)$.

En conclusion, on peut ramener la résolution d'une équation différentielle d'ordre n à la résolution de n équations différentielles couplées d'ordre 1. Ce résultat montre qu'il est essentiel de savoir calculer numériquement la solution d'une équation différentielle du premier ordre.

2.2 Solution formelle d'une équation différentielle d'ordre 1

On veut résoudre l'équation différentielle du premier ordre suivante :

$$\frac{dy}{dt} = f(t, y(t)). \tag{6}$$

En intégrant de a à t, il vient :

$$\int_{y(a)}^{y(t)} dy = \int_a^t f(u, y(u))du, \tag{7}$$

ou encore :

$$y(t) = y_0 + \int_a^t f(u, y(u))du, \tag{8}$$

où $y_0 \equiv y(a)$. Le problème avec cette solution formelle est que l'inconnue $y(t)$ se trouve sous l'intégrale.

2.3 Méthode d'Euler

On considère un échantillonnage régulier de N points t_n définis par :

$$t_n = a + \frac{b-a}{N-1}n, \quad \text{où } n \in \{0, 1, \ldots, N-1\}. \tag{9}$$

La grille de points comporte donc $N-1$ intervalles de longueur $h = \frac{b-a}{N-1}$.

On effectue un développement de Taylor de y à l'ordre 1 autour de t_n :

$$y(t_{n+1}) = y(t_n) + hy'(t_n) + O(h^2)$$

Par suite, on obtient :

$$y(t_{n+1}) = y(t_n) + hf(t_n, y(t_n)) + O(h^2) \tag{10}$$

en exploitant l'équation (6). L'équation précédente, à la base de la méthode d'Euler, permet d'obtenir la solution de l'équation différentielle (6) de proche en proche sur une grille régulière de points définis par l'équation (9). Cette méthode n'est pas très précise puisque l'erreur commise est $O(h^2)$.

2.4 Méthode de Runge-Kutta d'ordre 2

Effectuons un développement de Taylor de y à l'ordre 2 autour du point $\tilde{t} \equiv t_n + \frac{h}{2}$ au centre de l'intervalle $[t_n, t_{n+1}]$. Il vient alors :

$$\begin{aligned} y(t_n + h) &\equiv y(t_{n+1}) \\ &= y\left(t_n + \frac{h}{2}\right) + \frac{h}{2}y^{(1)}\left(t_n + \frac{h}{2}\right) + \frac{h^2}{8}y^{(2)}\left(t_n + \frac{h}{2}\right) + O(h^3), \end{aligned} \tag{6}$$

ou encore en posant $\tilde{y} = y\left(t_n + \frac{h}{2}\right)$:

$$y(t_{n+1}) = \tilde{y} + \frac{h}{2}f\left(t_n + \frac{h}{2}, \tilde{y}\right) + \frac{h^2}{8}y^{(2)}\left(t_n + \frac{h}{2}\right) + O(h^3). \tag{12}$$

Remplaçons $y^{(2)}\left(t_n + \frac{h}{2}\right)$ par son expression basée sur un schéma de différences finies :

$$y^{(2)}\left(t_n + \frac{h}{2}\right) = \frac{4}{h^2}[y(t_{n+1}) + y(t_n) - 2\tilde{y}] + O(h^2). \tag{13}$$

En substituant ce résultat dans l'équation (12), on obtient :

$$y(t_{n+1}) = \frac{h}{2}f\left(t_n + \frac{h}{2}, \tilde{y}\right) + \frac{1}{2}y(t_{n+1}) + \frac{1}{2}y(t_n) + O(h^3), \tag{14}$$

ou encore :

$$y(t_{n+1}) = y(t_n) + hf\left(t_n + \frac{h}{2}, \tilde{y}\right) + O(h^3). \tag{15}$$

Un développement de Taylor de y à l'ordre 1 autour de t_n conduit au résultat suivant :

$$\tilde{y} \equiv y\left(t_n + \frac{h}{2}\right) = y(t_n) + \frac{1}{2}hf(t_n, y(t_n)) + O(h^2), \tag{16}$$

où le terme $\frac{1}{2}hf(t_n, y(t_n))$ est noté k_1.

Avec cette approximation, on peut écrire :

$$hf\left(t_n + \frac{h}{2}, \tilde{y}\right) = hf\left(t_n + \frac{h}{2}, y(t_n) + \frac{k_1}{2}\right) + O(h^3), \tag{17}$$

où le terme $hf\left(t_n + \frac{h}{2}, y(t_n) + \frac{k_1}{2}\right)$ est noté k_2.

En insérant cette équation dans l'équation (15), on obtient finalement la formule de Runge-Kutta d'ordre 2 :

$$\begin{aligned} k_1 &= hf(t_n, y(t_n)), \\ k_2 &= hf\left(t_n + \frac{h}{2}, y(t_n) + \frac{k_1}{2}\right), \\ y(t_{n+1}) &= y(t_n) + k_2 + O(h^3). \end{aligned} \tag{18}$$

On note que l'erreur commise dans une méthode de type Runge-Kutta d'ordre 2 est $O(h^3)$ alors que l'erreur commise dans une méthode de type Euler est $O(h^2)$.

2.5 Méthode de Runge-Kutta d'ordre 4

La méthode de Runge-Kutta d'ordre 4 est de loin la plus utilisée. Sa dérivation, en comparaison de la méthode de Runge-Kutta d'ordre 2, est fastidieuse et ne sera pas exposée ici. Il n'en reste pas moins que cette méthode est facile à programmer et que les équations ont une forme relativement symétrique :

$$\begin{aligned} k_1 &= hf(t_n, y(t_n)), \\ k_2 &= hf\left(t_n + \frac{h}{2}, y(t_n) + \frac{k_1}{2}\right), \\ k_3 &= hf\left(t_n + \frac{h}{2}, y(t_n) + \frac{k_2}{2}\right), \\ k_4 &= hf(t_n + h, y(t_n) + k_3), \\ y(t_{n+1}) &= y(t_n) + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) + O(h^5). \end{aligned} \tag{19}$$

L'erreur commise avec cette méthode est $O(h^5)$. Elle est très robuste et utilisée dans la plupart des fonctions permettant de résoudre des équations différentielles.

3 Méthodes numériques et programmes

3.1 Résolution numérique d'une équation différentielle

1. Écrire une fonction, qu'on note `euler(f, a, b, y0, N)`, permettant d'intégrer une équation différentielle de la forme :

$$y' = f(t, y) \quad \text{avec} \quad y(a) = y_0 \tag{20}$$

sur l'intervalle $[a, b]$ avec la méthode d'Euler décrite dans la partie 1.3. Écrire un programme `ed.py` exploitant cette fonction afin de résoudre

$$y' + y = t$$

sur l'intervalle $[0, 2]$ avec la condition aux limites $y_0 \equiv y(0) = 1$. Il nous faut déterminer en particulier la valeur de N permettant d'obtenir moins de 1% d'erreur en relatif sur la solution de l'équation différentielle. Ainsi, nous allons représenter sur un même graphique la solution numérique et la solution analytique.

Le programme `ed.py` doit commencer par :

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

3.2 Code Python

Le code Python pour résoudre l'équation différentielle est le suivant :

```python
def euler(f, a, b, y0, N):
    h = (b - a) / (N - 1)
    t = np.linspace(a, b, N)
    y = np.zeros(N)
    y[0] = y0
    for i in range(1, N):
        y[i] = y[i-1] + h * f(t[i-1], y[i-1])
    return t, y

def f(t, y):
    return t - y

def analytical_solution(t):
    return t - 1 + 2 * np.exp(-t)

a, b = 0, 2
y0 = 1
N = 100

t, y_euler = euler(f, a, b, y0, N)
y_analytical = analytical_solution(t)

plt.plot(t, y_euler, label='Euler')
plt.plot(t, y_analytical, label='Solution Analytique', linestyle='dashed')
plt.xlabel('t')
plt.ylabel('y(t)')
plt.legend()
plt.show()

# Calcul de l'erreur relative
error = np.abs((y_euler - y_analytical) / y_analytical)
max_error = np.max(error)
print(f'Max error: {max_error:.2%}')
```

Voici donc la solution numérique et la solution analytique trouvé et leurs comparaison dans un graphique dans la figure 1. Par ailleurs, la valeurs de N trouvée est de 100.

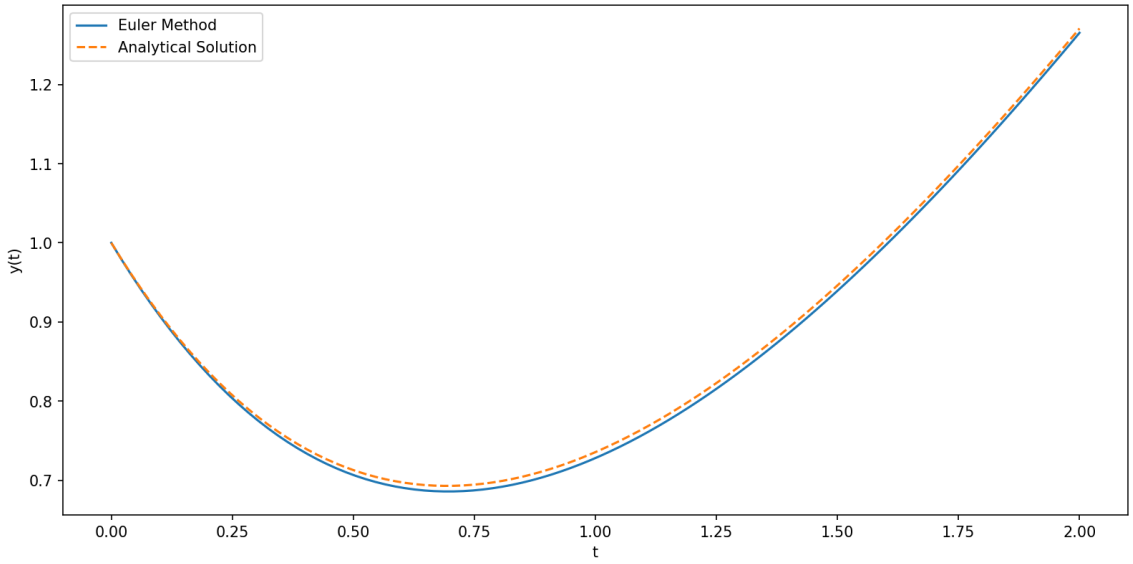

FIGURE 1 – Solution analytique et méthode d'Euler.

3.3 Runge Kutta 2 et 4

Dans ce script, les fonctions rk2 et rk4 implémentent respectivement les méthodes de Runge-Kutta d'ordre 2 et 4. La solution analytique est calculée avec la fonction analyticalsolution. Les solutions obtenues avec les méthodes numériques et la solution analytique sont tracées sur le même graphique.

L'erreur relative pour chaque méthode est calculée et affichée. Vous pouvez ajuster la valeur de N pour voir comment l'erreur évolue avec le nombre de points. Nous pouvons donc implémenter le programme suivant :

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

def f(y, t):
    return t - y

def rk2(f, a, b, y0, N):
    h = (b - a) / (N - 1)
    t = np.linspace(a, b, N)
    y = np.zeros(N)
    y[0] = y0
    for i in range(1, N):
        k1 = h * f(y[i-1],t[i-1])
        k2 = h * f( y[i-1] + k1/2,t[i-1] + h/2)
        y[i] = y[i-1] + k2
    return t, y

def rk4(f, a, b, y0, N):
    h = (b - a) / (N - 1)
    t = np.linspace(a, b, N)
    y = np.zeros(N)
    y[0] = y0
    for i in range(1, N):
        k1 = h * f(y[i-1],t[i-1])
        k2 = h * f( y[i-1] + k1/2,t[i-1] + h/2)
        k3 = h * f( y[i-1] + k2/2,t[i-1] + h/2)
        k4 = h * f( y[i-1] + k3, t[i-1] + h)
        y[i] = y[i-1] + (k1 + 2*k2 + 2*k3 + k4) / 6
    return t, y

def analytical_solution(t):
    return t - 1 + 2 * np.exp(-t)

a, b = 0, 2
y0 = 1
N = 10

t, y_rk2 = rk2(f, a, b, y0, N)
t, y_rk4 = rk4(f, a, b, y0, N)
y_odeint = odeint(f, y0, t).flatten()
y_analytical = analytical_solution(t)

plt.plot(t, y_rk2, label='Runge-Kutta 2')
plt.plot(t, y_rk4, label='Runge-Kutta 4')
plt.plot(t, y_odeint, label='odeint', linestyle='dashed')
plt.plot(t, y_analytical, label='Analytical Solution', linestyle='dotted')
plt.xlabel('t')
plt.ylabel('y(t)')
plt.legend()
plt.show()

# Calcul de l'erreur relative
error_rk2 = np.abs((y_rk2 - y_analytical) / y_analytical)
```

```
error_rk4 = np.abs((y_rk4 - y_analytical) / y_analytical)
error_odeint = np.abs((y_odeint - y_analytical) / y_analytical)

print(f'Max error RK2: {np.max(error_rk2):.2%}')
print(f'Max error RK4: {np.max(error_rk4):.2%}')
print(f'Max relative error odeint: {np.max(error_odeint):.2%}')
```

3.4 Résultats

Nous pouvons maintenant analyser les résultats et les comparer à la solution analytique et aux solutions issu de Runge Kutta. Pour cela, nous avons zoomer pour voir les échelles des érreurs. De même, avec N = 10, on a une erreur maximum relative pour RK2 : 1.00 , pour RK4 : 0.00et pour odeint : 0.00Nous pouvons donc conclure que la méthode Odeint est la plus précise même s'il y a une legère erreur avec la solution analytique. De même, Runge Kutta est une approximation qui peut être améliorer en augmentant l'ordre de ses polynomes.

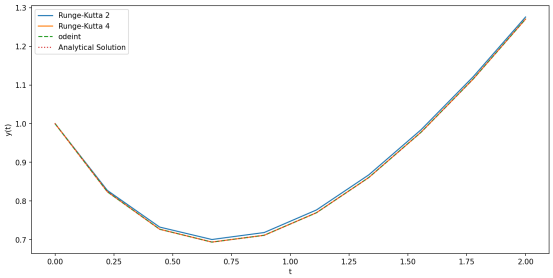

FIGURE 2 – Solutions Runge Kutta 2 et 4, odeint et analytique.

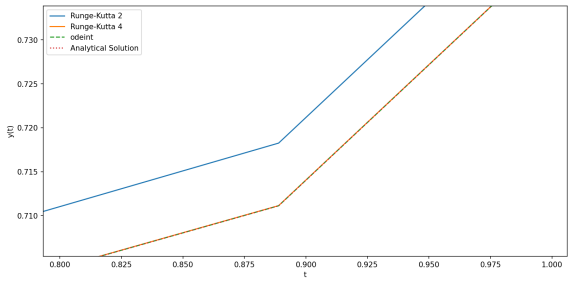

FIGURE 3 – Solutions Runge Kutta 2 et 4, odeint et analytique.

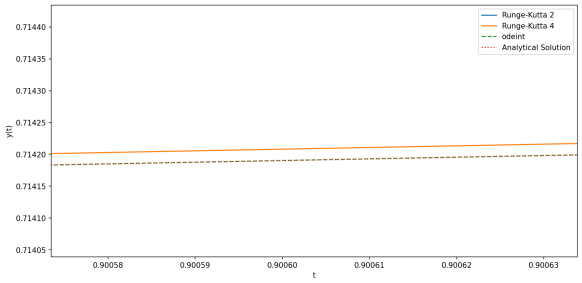

FIGURE 4 – Solutions Runge Kutta 2 et 4, odeint et analytique.

3.5 Pendule simple

On considère un pendule pesant de longueur l dont le mouvement est régi par l'équation :

$$\ddot{\theta} + \frac{g}{l}\sin(\theta) = 0 \quad (21) \tag{7}$$

où g représente l'intensité du champ de pesanteur et θ l'angle entre la verticale et le pendule. Montrer en vous inspirant de la partie 1.1 que la résolution de l'équation différentielle (21) peut se ramener à la résolution de deux équations différentielles couplées d'ordre 1 à mettre sous la forme :

$$\frac{dy_1}{dt} = f_1(t, y_1, y_2) \tag{8}$$

$$\frac{dy_2}{dt} = f_2(t, y_1, y_2) \quad (22) \tag{9}$$

où les fonctions $f_i(t, y_1, y_2)$ sont à préciser pour $i = 1, 2$. Écrire un programme, qu'on note `oscillator.py`, exploitant une méthode de Runge-Kutta d'ordre 4 pour résoudre l'équation différentielle (21). Prendre $l = 9.81$ cm et représenter sur une figure la solution calculée sur un intervalle de temps $[a, b]$ judicieusement choisi pour les conditions initiales suivantes : $\theta(0) = \theta_0 = 10°$ et $\dot{\theta}(0) = 0$. Superposer sur la figure la solution analytique.

```python
import numpy as np
import matplotlib.pyplot as plt

def runge_kutta_4(f1, f2, y1_0, y2_0, t):
    h = t[1] - t[0]
    y1 = [y1_0]
    y2 = [y2_0]
    for i in range(len(t)-1):
        k1_1 = h * f1(y1[-1], y2[-1])
        k1_2 = h * f2(y1[-1], y2[-1])

        k2_1 = h * f1(y1[-1] + k1_1/2, y2[-1] + k1_2/2)
        k2_2 = h * f2(y1[-1] + k1_1/2, y2[-1] + k1_2/2)

        k3_1 = h * f1(y1[-1] + k2_1/2, y2[-1] + k2_2/2)
        k3_2 = h * f2(y1[-1] + k2_1/2, y2[-1] + k2_2/2)

        k4_1 = h * f1(y1[-1] + k3_1, y2[-1] + k3_2)
        k4_2 = h * f2(y1[-1] + k3_1, y2[-1] + k3_2)

        y1.append(y1[-1] + (k1_1 + 2*k2_1 + 2*k3_1 + k4_1)/6)
        y2.append(y2[-1] + (k1_2 + 2*k2_2 + 2*k3_2 + k4_2)/6)

    return y1, y2

# Definitions
g = 9.81
l = 9.81 / 100
f1 = lambda y1, y2: y2
f2 = lambda y1, y2: -g/l * np.sin(y1)
y1_0 = np.deg2rad(10)
y2_0 = 0
t = np.linspace(0, 10, 1000)

y1, y2 = runge_kutta_4(f1, f2, y1_0, y2_0, t)

theta_analytical = y1_0 * np.cos(np.sqrt(g/l) * t)
# Plot
plt.plot(t, y1, label='RK4 Solution')
plt.plot(t, theta_analytical, label='Analytical Solution', linestyle='--')
plt.xlabel('Time (s)')
plt.ylabel('  (t) (radians)')
plt.legend()
plt.show()
```

Les résultats sont assez satisfaisant et peut observer ainsi, l'erreur en zoomant :

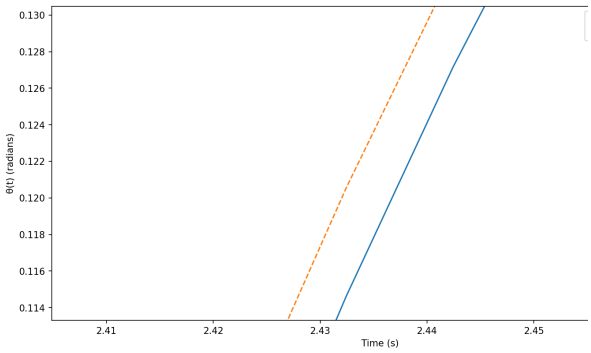

FIGURE 5 – Solution analytique et Runge Kutta.

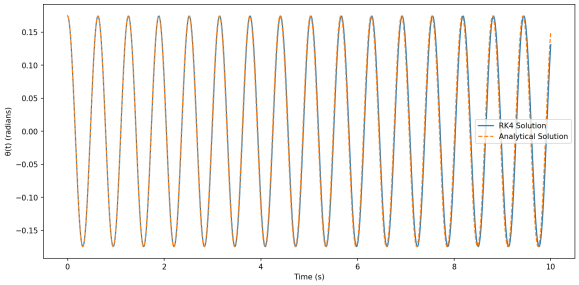

FIGURE 6 – Runge Kutta et analytique .

3.6 Démonstration de la période des oscillations pour un pendule

L'équation du mouvement pour un pendule simple est :

$$\frac{d^2\theta}{dt^2} + \frac{g}{l}\sin(\theta) = 0 \tag{10}$$

On multiplie par $\frac{d\theta}{dt}$

$$\frac{d\theta}{dt}\frac{d^2\theta}{dt^2} + \frac{g}{l}\sin(\theta)\frac{d\theta}{dt} = 0 \tag{11}$$

On intègre par rapport à t

$$\frac{1}{2}\left(\frac{d\theta}{dt}\right)^2 = \frac{g}{l}\cos(\theta) + E$$

où E est la constante d'intégration. En utilisant les conditions initiales :

$$E = -\frac{g}{l}\cos(\theta_0) \tag{12}$$

Exprimer dt

$$dt = \frac{d\theta}{\sqrt{2\left(\frac{g}{l}\cos(\theta) - \frac{g}{l}\cos(\theta_0)\right)}}$$

On effectue un changement de variable
En utilisant $k = \sin(\theta_0/2)$ et $d\phi = \frac{1}{2}d\theta$:

$$dt = \frac{2d\phi}{\sqrt{\frac{4g}{l}\sin(\phi)(1 - \sin(\phi)k^2)}}$$

On trouve T

$$T = 4\int_0^{\pi/2} \frac{2d\phi}{\sqrt{\frac{4g}{l}\sin(\phi)(1 - \sin(\phi)k^2)}}$$

On trouve la période des petites oscillations

$$T_0 = 2\pi\sqrt{\frac{l}{g}}$$

Par rapport des périodes, on a :

$$\frac{T}{T_0} = \frac{2}{\pi}\int_0^{\pi/2} \frac{d\phi}{\sqrt{1 - k^2\sin^2(\phi)}}$$

Nous pouvons maintenant faire le programme python afin d'évaluer les oscillations et le repport des oscillations.

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

```
# Constantes
g = 9.81
l = 1.0  # Pour l'exemple, prenons 1 m tre comme longueur du pendule
T0 = 2 * np.pi * np.sqrt(l / g)

# Int grand de l' quation  donn e
def integrand(phi, k):
    return 1 / np.sqrt(1 - k**2 * np.sin(phi)**2)

# Calculer la p riode T en fonction de theta_0
def compute_T(theta_0):
    k = np.sin(theta_0 / 2)
    integral_value, _ = quad(integrand, 0, np.pi/2, args=(k))
    return 2 * np.pi * integral_value / T0

# Tracer T/T0 en fonction de theta_0
theta_0_values = np.linspace(0.01, np.pi - 0.01, 400)  #  viter  les singularit s  0 et pi
T_values = [compute_T(theta_0) for theta_0 in theta_0_values]

plt.plot(theta_0_values, T_values)
plt.xlabel('$\\theta_0$ (radians)')
plt.ylabel('$T/T_0$')
plt.title('Rapport des p riodes en fonction de $\\theta_0$')
plt.grid(True)
plt.show()
```

En résultat, nous pouvons mesurer le rapport des périodes :

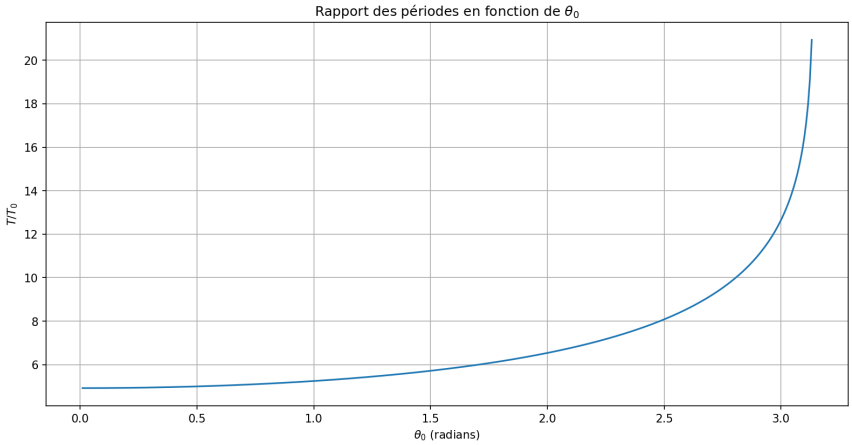

FIGURE 7 – Rapports des périodes

4 Conclusion

Au terme de ce travail pratique, nous avons pu appréhender et mettre en œuvre diverses méthodes mathématiques et numériques pour la résolution d'équations différentielles. Nous avons transformé des équations différentielles d'ordre supérieur en systèmes d'équations du premier ordre, facilitant ainsi leur analyse et leur résolution numérique.

Les méthodes d'Euler et de Runge-Kutta, de différents ordres, ont été explorées et appliquées avec succès, permettant de comprendre leurs avantages et limites respectives. La méthode d'Euler, bien que simple à implémenter, a montré ses limites en termes de précision, particulièrement pour des pas de temps plus grands. D'un autre côté, les méthodes de Runge-Kutta, en particulier celle d'ordre 4, ont démontré une capacité supérieure à fournir des solutions précises, même avec des pas de temps relativement grands.

Les programmes Python développés ont joué un rôle clé dans la visualisation et la comparaison des résultats obtenus par ces différentes méthodes, offrant une perspective concrète sur leur performance et leur fiabilité. L'analyse des erreurs relatives a également été un aspect crucial, permettant de quantifier la précision des solutions numériques par rapport aux solutions analytiques.