Metody Rungego-Kutty

Karol Janik, Michał Kazanecki

23 grudnia 2020

1 Wstęp

Równanie postaci:

$$y^{(n)} = f(x, y, y', ..., y^{(n-1)}) \tag{1}$$

nazywamy równaniem różniczkowym zwyczajnym rzędu n. Równanie to można przekształcić w układ n równań rzędu 1:

$$y_0' = y_1, \quad y_1' = y_2, \quad ... \quad y_n' = f(x, y_0, y_1, ..., y_{n-1}) \tag{2}$$

Gdzie:

$$y_0 = y, \quad y_1 = y', \quad y_2 = y'', \quad ... \quad y_{n-1} = y^{(n-1)} \tag{3}$$

W postaci wektorowej powyższe równania można zapisać jako:

$$\mathbf{y}' = \mathbf{F}(x, \mathbf{y}) \tag{4}$$

Gdzie:

$$\mathbf{F}(x, \mathbf{y}) = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ f(x, \mathbf{y}) \end{pmatrix} \tag{5}$$

Rozwiązanie takiego układu wymaga znajomości n warunków zewnętrznych. Gdy warunki te dane są dla tej samej wartości x, problem ten nazywamy problemem zagadnienia początkowego. Numeryczne rozwiązanie można osiągnąć, na przykład metodą Eulera, która wykorzystuje rozwinięcie funkcji w szereg Taylora do wyrazów pierwszego rzędu:

$$\mathbf{y}(x + h) \approx \mathbf{y}(x) + \mathbf{y}'(x)h \tag{6}$$

Metoda ta jest jednak obarczona dużym błędem (błąd skumulowany jest tu rzędu $\mathcal{O}(h)$) i łatwo traci stabilność. Wymaga więc bardzo małych kroków h w celu zachowania dokładności, to z kolei prowadzi do konieczności wykonania wielu obliczeń. Z tego powodu w praktyce stosuje się bardziej zaawansowane metody jak na przykład metody Rungego-Kutty, którym poświęcona jest ta praca.

2 Metody Rungego-Kutty

Sekcja ta zawiera przedstawienie metod Rungego-Kutty rzędu 2, 4 oraz z krokiem adaptacyjnym. W dodatkach znajduje się ich implementacja w języku Python.

2.1 Metoda Rungego-Kutty rzędu 2

W celu poprawy dokładnośći całkowania numerycznego można użyć rozwinięcia Taylora zachowując większą ilośc wyrazów. Prowadzi to jednak do konieczności obliczenia pochodnych wyższych rzędów. Metoda Rungego-Kutty rzędu 2 omija ten problem, poprzez rozwinięcie funkci w szereg Taylora wokół środkowego punktu przedziału całkowania:

$$f(t,y) \approx f(t_{n+1/2}, y_{n+1/2}) + (t - t_{n+1/2})\frac{df}{dx}(t_{n+1/2}) \tag{7}$$

Podczas całkowania drugi wyraz znika, co prowadzi do wzoru:

$$y_{n+1} \approx y_n + hf(t_{n+1/2}, y_{n+1/2}) \tag{8}$$

Wartość $y_{n+1/2}$ można obliczyć metodą Eulera:

$$y_{n+1/2} \approx y_n + \frac{1}{2}hf(t_n, y_n) \tag{9}$$

Ostatecznie algorytm przybiera postać:

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \mathbf{k}_2 \tag{10}$$

$$\mathbf{k}_2 = h\mathbf{f}\left(t_n + \frac{h}{2}, \mathbf{y}_n + \frac{\mathbf{k}_1}{2}\right), \qquad \mathbf{k}_1 = h\mathbf{f}(t_n, \mathbf{y}_n) \tag{11}$$

2.2 Metoda Rungego-Kutty rzędu 4

Wyprowadzenie metody Rungego-Kutty rzędu 4 jest znacznie bardziej skomplikowane (znaleźć je można w [2]). Metoda ta wykorzystuje rozwinięcie funkcji w szereg Taylora wokół środkowego punktu przedziału do wyrazów drugiego rzędu. Prowadzi to do następujących wzorów:

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{1}{6}\left(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4\right) \tag{12}$$

$$\mathbf{k}_1 = h\mathbf{f}(t_n, \mathbf{y}_n) \qquad \mathbf{k}_2 = h\mathbf{f}\left(t_n + \frac{h}{2}, \mathbf{y}_n + \frac{\mathbf{k}_1}{2}\right)$$

$$\mathbf{k}_3 = h\mathbf{f}\left(t_n + \frac{h}{2}, \mathbf{y}_n + \frac{\mathbf{k}_2}{2}\right) \qquad \mathbf{k}_4 = h\mathbf{f}\left(t_n + h, \mathbf{y}_n + k_3\right)$$

2.3 Metoda Rungego-Kutty z krokiem adaptacyjnym

Metoda z krokiem adaptacyjnym pozwala na automatyczne wyznaczanie kroku w celu utrzymania błędu na określonym poziomie. Jest to robione z wykorzystaniem metod rzędu m i $m+1$. Oszacowanie błędu metody stopnia m uzyskuje się ze wzoru:

$$\mathbf{E}(h) = \mathbf{y}_{m+1}(x+h) - \mathbf{y}_m(x+h) \tag{13}$$

Metody te mają wspólne punkty, w których oblicza się $\mathbf{F}(x, \mathbf{y})$, dzięki czemu oszacowanie błędu wiąże się ze znacznie mniejszym kosztem obliczeniowym niż użycie metody wyższego rzędu. W wykorzystanym przez nas algorytmie użyta została metoda 4 i 5 rzędu. Krok jest automatycznie podwajany. Jeśli błąd utrzymuje się w akceptowalnych granicach, krok ten jest wykorzystywany aby przyspieszyć działanie algorytmu. Jeśli nie, następuje zmniejszenie kroku aż błąd osiągnie akceptowalną wartość. Więcej o tej metodzie przeczytać można [2] w lub [3].

3 Przebieg pracy i wyniki

Pierwszym etapem pracy było zaimplementowanie metody drugiego rzędu i użycie jej do rozwiązania przykładowego równania. Na rysunku 1 zobaczyc można rozwiązanie równania:

$$x'' + 6x^5 = 0, \qquad v(0) = x'(0) = 0, \qquad x(0) = 1 \tag{14}$$

Utożsamiając drugą pochodną z siłą działajacą na cząstkę, jest to równanie dla potencjału opisanego wielomianem 6-tego stopnia. Funkcja taka bardzo szybko rośnie, więc zachowanie cząstki powinno być zbliżone do ruchu w nieskończonej studni potencjału. Rysunki odzwierciedlają to przypuszczenie.

Na rysunkach 2-5 przedstawione zostały rozwiązania metodą drugiego rzędu o różnej długości kroku, dla równania oscylatora harmonicznego:

$$x'' + \omega^2 x = 0, \qquad v(0) = x'(0) = 1, \qquad x(0) = 0, \qquad \omega = \pi \tag{15}$$

Rozwiązaniem tego równania jest funkcja $x = sin(\pi t)$ o okresie $T = 2$. Na rysunkach, dla porównania, zostały umieszczone rozwiązania analityczne. Widać, że dla dużych wartości kroku rozwiązania są niestabilne. Wraz ze zmniejszeniem kroku rozwiązanie numeryczne przybliża się do analitycznego. Rysunek 6 przedstawia rozwiązanie tego samego równania przy wartości początkowej $v(0) = 2$. Widać, że okres oscylatora jest niezależny od warunków początkowych.

W tabeli 1 przedstawione zostało porównanie metod drugiego i czwartego rzedu oraz metody z krokiem adaptacyjnym. Błąd liczony jest według wzoru:

$$\frac{\Delta E}{E} = \frac{\omega^2 x^2 + v^2 - v_0^2}{v_0^2} \tag{16}$$

i uśredniony po wszystkich krokach. Interpretować to można jako względne odchylenie od stałej wartości energii oscylatora. Kroki zostały dobrane metodą

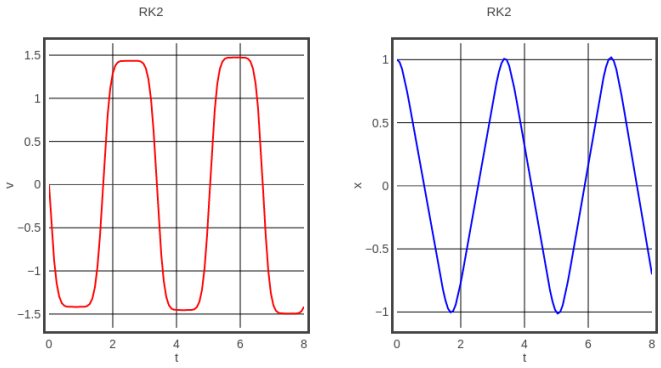
Metoda	Krok pocz.	Iteracje	Błąd
rk2	0.004	2001	$6.24 \cdot 10^{-6}$
rk4	0.0107	751	$5.55 \cdot 10^{-6}$
rk45	1.6	166	$6.26 \cdot 10^{-6}$

Tabela 1: Porównanie metod Rungego-Kutty

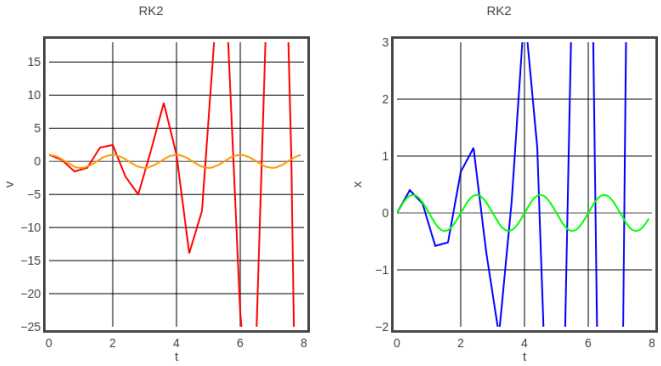
prób i błędów tak, aby błąd względny był tego samego rzędu. Wykresy dla pozostałych metod nie wnoszą nic nowego, więc zostały pominięte.

4 Podsumowanie

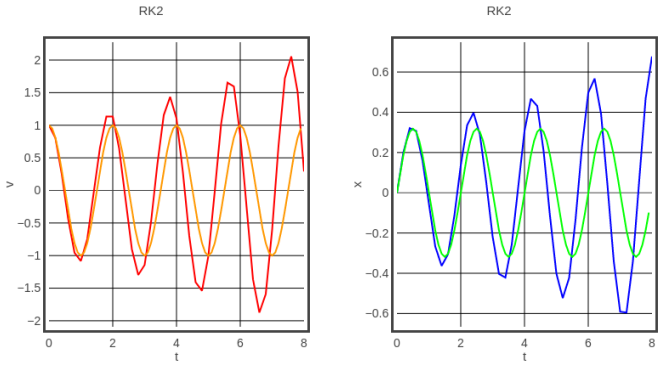
W pracy udało się skutecznie zaimplementować metody Rungego-Kutty 2 i 4 rzędu i metodę z krokiem adaptacyjnym. Ukazany został wzrost dokładności rozwiązania przy malejącym kroku. Widać też, że metoda z krokiem adaptacyjnym daje rozwiązanie z błędem tego samego rzędu jak inne metody przy najmniejszej liczbie iteracji.



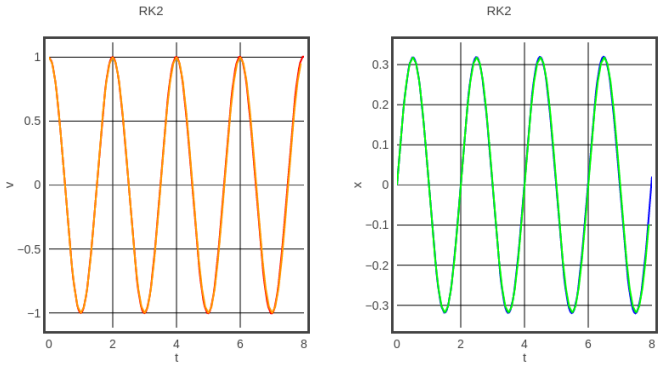
Rysunek 1: Rozwiązanie równania $x'' + 6x^5 = 0$ metodą Rungego-Kutty 2 rzędu



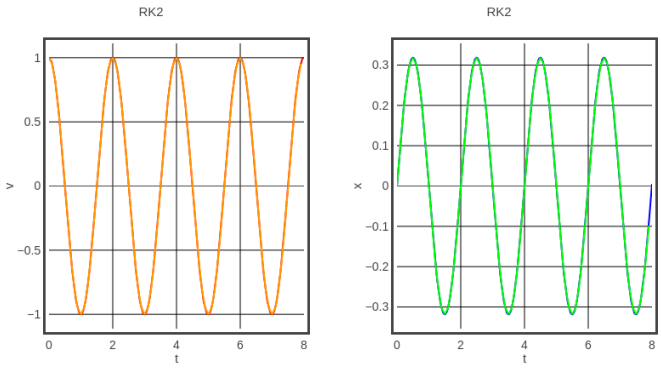
Rysunek 2: Rozwiązanie równania $x'' + \pi^2 x = 0$ metodą Rungego-Kutty 2 rzędu, h=T/5



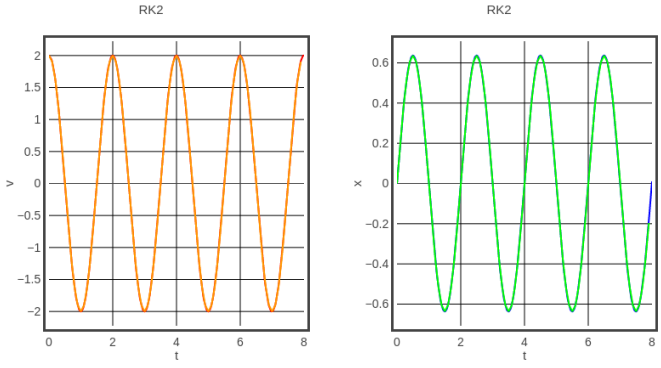
Rysunek 3: Rozwiązanie równania $x'' + \pi^2 x = 0$ metodą Rungego-Kutty 2 rzędu, h=T/10



Rysunek 4: Rozwiązanie równania $x'' + \pi^2 x = 0$ metodą Rungego-Kutty 2 rzędu, h=T/50



Rysunek 5: Rozwiązanie równania $x'' + \pi^2 x = 0$ metodą Rungego-Kutty 2 rzędu, h=T/100



Rysunek 6: Rozwiązanie równania $x'' + \pi^2 x = 0$ metodą Rungego-Kutty 2 rzędu, h=T/100, v(0)=2

A RK2

```
#!/usr/bin/env python3

from vpython import *
from numpy import zeros
from time import time

a = 0.
b = 8.
#n = 100
flops = 0
ydumb = zeros((2), float)
y = zeros((2), float)
fReturn = zeros((2), float)
y[0] = 0.
y[1] = 1.
t = a
T = 2.
omega = 2*pi/T
A = y[1]/omega
h = T/500

print("Init. h: ", h)
err = 0.
sum_err = 0.
E = 0.
Eexact = y[1]*y[1]

def f(t, y, fReturn):
    fReturn[0] = y[1]
    fReturn[1] = -omega*omega*y[0]

def rk2Function(h, fReturn):
    k1 = zeros((2), float)
    k2 = zeros((2), float)
    k1[0] = h * fReturn[0]; k1[1] = h *fReturn[1]
    for i in range(0, 2): ydumb[i] = y[i] + k1[i]/2.
    f(t + h/2., ydumb, fReturn)
    k2[0] = h * fReturn[0]; k2[1] = h * fReturn[1]
    for i in range(0, 2): y[i] = y[i] + k2[i]

graph1 = graph(x=0, y=0, width = 400, height = 400, title = 'RK2',
xtitle = 't', ytitle = 'x', xmin=0, xmax=8, ymin=-2, ymax=3, fast=False)
funct1 = gcurve(color = color.blue)
funct1_a = gcurve(color = color.green)
```

```
graph2 = graph(x=400, y=0, width = 400, height = 400, title = 'RK2',
xtitle = 't', ytitle = 'v', xmin=0, xmax=8, ymin=-25, ymax=18, fast=False)
funct2 = gcurve(color = color.red)
funct2_a = gcurve(color = color.orange)
funct1.plot(pos = (t, y[0]))
funct2.plot(pos = (t, y[1]))
for t1 in range(0, 80):
    funct1_a.plot(t1/10, A*sin(omega*t1/10))
    funct2_a.plot(t1/10, A*omega*cos(omega*t1/10))

sum_time = 0.

while (t < b):
    if ((t+h) > b): h = b - t
    f(t, y, fReturn)
    start = time()
    rk2Function(h, fReturn)
    t = t + h
    flops = flops + 1
    E = omega*omega*y[0]*y[0] + y[1]*y[1]
    err = abs((E - Eexact)/Eexact)
    sum_err += err
    #print(E, Eexact)
    stop = time()
    sum_time += stop - start
    funct1.plot(pos = (t, y[0]))
    funct2.plot(pos = (t, y[1]))

print("time: ", sum_time)
print("error: ", sum_err/flops)
print("flops: ", flops)
```

B RK4

```
#!/usr/bin/env python3

from vpython import *
from numpy import zeros
from time import time

a = 0.
b = 8.
n = 750
flops = 0
ydumb = zeros((2), float)
```

```
y = zeros((2), float)
fReturn = zeros((2), float)
y[0] = 0.
y[1] = 1.
t = a
T = 2.
omega = 2*pi/T
h = (b-a)/n

print("Init. h: ", h)
error = 0.
sum_err = 0.
E = 0.
Eexact = y[1]*y[1]

def f(t, y, fReturn):
    fReturn[0] = y[1]
    fReturn[1] = -omega*omega*y[0]

def rk4Function(h, fReturn):
    k1 = zeros((2), float)
    k2 = zeros((2), float)
    k3 = zeros((2), float)
    k4 = zeros((2), float)
    k1[0] = h * fReturn[0]; k1[1] = h *fReturn[1]
    for i in range(0, 2): ydumb[i] = y[i] + k1[i]/2.
    f(t + h/2., ydumb, fReturn)
    k2[0] = h * fReturn[0]; k2[1] = h * fReturn[1]
    for i in range(0, 2): ydumb[i] = y[i] + k2[i]/2.
    f(t + h/2., ydumb, fReturn)
    k3[0] = h * fReturn[0]; k3[1] = h * fReturn[1]
    for i in range(0, 2): ydumb[i] = y[i] + k3[i]
    f(t + h, ydumb, fReturn)
    k4[0] = h * fReturn[0]; k4[1] = h * fReturn[1]
    for i in range(0, 2): y[i] = y[i] + (k1[i]+2.*(k2[i]+k3[i])+k4[i])/6.

graph1 = graph(x=0, y=0, width = 400, height = 400, title = 'RK4',
xtitle = 't', ytitle = 'Y[0]', xmin=0, xmax=8, ymin=-2, ymax=3)
funct1 = gcurve(color = color.blue)
graph2 = graph(x=400, y=0, width = 400, height = 400, title = 'RK4',
xtitle = 't', ytitle = 'Y[1]', xmin=0, xmax=8, ymin=-25, ymax=18)
funct2 = gcurve(color = color.red)
funct1.plot(pos = (t, y[0]))
funct2.plot(pos = (t, y[1]))

sum_time = 0.
```

```
while (t < b):
    if ((t+h) > b): h = b - t
    f(t, y, fReturn)
    start = time()
    rk4Function(h, fReturn)
    t = t + h
    flops = flops + 1
    E = omega*omega*y[0]*y[0] + y[1]*y[1]
    error = abs((E - Eexact)/Eexact)
    sum_err += error
    stop = time()
    sum_time += stop - start
    funct1.plot(pos = (t, y[0]))
    funct2.plot(pos = (t, y[1]))

print("time: ", sum_time)
print("error: ", sum_err)
print("flops: ", flops)
```

C RK45

```
 #!/usr/bin/env python3

from vpython import *
from numpy import zeros
from time import time

a = 0.
b = 8.
n = 5
flops = 0
ydumb = zeros((2), float)
y = zeros((2), float)
fReturn = zeros((2), float)
y[0] = 0.
y[1] = 1.
t = a
T = 2.
omega = 2*pi/T
h = (b-a)/n

print("Init. h: ", h)

def f(t, y, fReturn):
```

```
    fReturn[0] = y[1]
    fReturn[1] = -omega*omega*y[0]

Tol = 1.0E-8
err = zeros((2), float)
hmin = h/64; hmax = h * 64

Eexact = y[1]*y[1]; sum_err = 0.

def rk45Function(fReturn):
    global t
    global h
    global flops
    global sum_err
    error = 0.;

    k1 = zeros((2), float)
    k2 = zeros((2), float)
    k3 = zeros((2), float)
    k4 = zeros((2), float)
    k5 = zeros((2), float)
    k6 = zeros((2), float)
    k1[0] = h * fReturn[0]; k1[1] = h * fReturn[1]
    for i in range(2):
        ydumb[i] = y[i] + k1[i]/4
    f(t + h/4, ydumb, fReturn)
    k2[0] = h * fReturn[0]; k2[1] = h * fReturn[1]
    for i in range(2):
        ydumb[i] = y[i] + 3*k1[i]/32 + 9*k2[i]/32
    f(t + 3*h/8, ydumb, fReturn)
    k3[0] = h * fReturn[0]; k3[1] = h * fReturn[1]
    for i in range(2):
        ydumb[i] = y[i] + 1932*k1[i]/2197 - 7200*k2[i]/2197 + 7296*k3[i]/2197
    f(t + 12*h/13, ydumb, fReturn)
    k4[0] = h * fReturn[0]; k4[1] = h * fReturn[1]
    for i in range(2):
        ydumb[i] = y[i] + 439*k1[i]/216 - 8*k2[i]
        + 3680*k3[i]/513 - 845*k4[i]/4104
    f(t + h, ydumb, fReturn)
    k5[0] = h * fReturn[0]; k5[1] = h * fReturn[1]
    for i in range(2):
        ydumb[i] = y[i] - 8*k1[i]/27 + 2*k2[i]
        - 3544*k3[i]/2565 + 1895*k4[i]/4104 - 11*k5[i]/40
    f(t + h/2, ydumb, fReturn)
    k6[0] = h * fReturn[0]; k6[1] = h * fReturn[1]
    for i in range(2):
```

```
        err[i] = abs(k1[i]/360 - 128*k3[i]/4275
        - 2197*k4[i]/75240 + k5[i]/50 + 2*k6[i]/55)
    if (err[0] < Tol or err[1] < Tol or h <= 2*hmin):
        for i in range(2):
            y[i] = y[i] + 25*k1[i]/216 + 1408*k3[i]/2565
            + 2197*k4[i]/4104 - k5[i]/5
        t = t + h

    if (err[0] == 0 or err[1] == 0):
        s = 0
    else:
        s = 0.84*pow(Tol*h/err[0], 0.25)
    if (s < 0.75 and h > 2*hmin):
        h = h/2
    elif(s > 1.5 and 2*h < hmax):
        h = h*2
    flops = flops + 1
    E = omega*omega*y[0]*y[0] + y[1]*y[1]
    error = abs((E - Eexact)/Eexact)
    sum_err += error

graph3 = graph(x=0, y=0, width = 400, height = 400, title = 'RK45',
xtitle = 't', ytitle = 'Y[0]', xmin=0, xmax=8, ymin=-2, ymax=3, fast=False)
funct3 = gcurve(color = color.blue)
graph4 = graph(x=400, y=0, width = 400, height = 400, title = 'RK45',
xtitle = 't', ytitle = 'Y[1]', xmin=0, xmax=8, ymin=-25, ymax=18, fast=False)
funct4 = gcurve(color = color.red)
funct3.plot(pos = (t, y[0]))
funct4.plot(pos = (t, y[1]))

sum_time = 0.

while (t < b):
    if ((t + h) > b): h = b - t
    f(t, y, fReturn)
    start = time()
    rk45Function(fReturn)
    stop = time()
    funct3.plot(pos = (t, y[0]))
    funct4.plot(pos = (t, y[1]))
    sum_time += stop - start

print("error: ", sum_err/flops)
print("time: ", sum_time)
print("flops: ", flops)
```

Literatura

[1] Landau R. H., Paez M. J., Bordeinau C. C., *A Survey of Computational Physics*, Princeton University Press 2012

[2] https://en.wikipedia.org/wiki/Runge-Kutta_methods (Dostęp 05.11.2020)

[3] Kiusalaas J, *Numerical methods in engineering with Python 3*, Cambridge University Press 2013