Laboratorium 10

Rozwiązywanie równań różniczkowych zwyczajnych

Bartłomiej Szubiak

21.05.2024

Zad1:

Dane jest równanie różniczkowe (zagadnienie początkowe): $y' + y \cos x = \sin x \cos x$, $y(0) = 0$
Znaleźć rozwiązanie metodą Rungego-Kutty i metodą Eulera.
Porównać otrzymane rozwiązanie z rozwiązaniem dokładnym $y(x) = e^{-\sin x} + \sin x - 1$.

Metoda Rungego-Kutty opiera się na wzorze rekurencyjnym:

$$\begin{cases} y_{n+1} = y_n + \Delta y_n \\ \Delta y_n = \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4), \end{cases}$$

$$\begin{cases} k_1 = hf(x_n, y_n) \\ k_2 = hf(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}) \\ k_3 = hf(x_n + \frac{h}{2}, y_n + \frac{k_2}{2}) \\ k_4 = hf(x_n + h, y_n + k_3). \end{cases}$$

Metoda Eulera to szczególny przypadek tej metody, gdzie:

$$y_{n+1} = y_n + k_1.$$

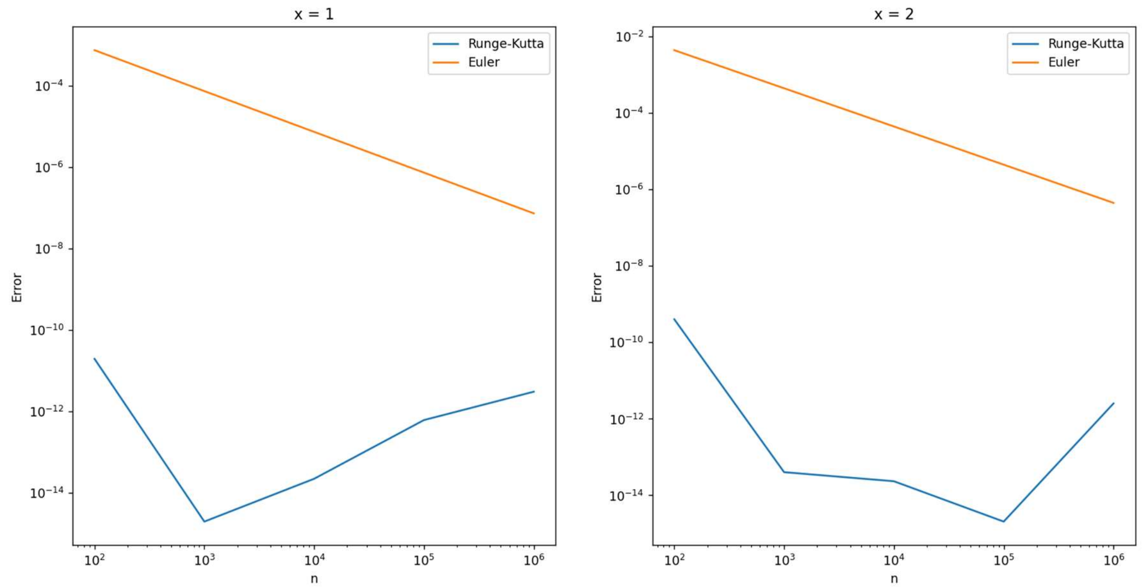
Użyty kod języka Python:

```
from math import sin, cos, e

def f(x, y):
    return sin(x) * cos(x) - y * cos(x)

def exact_solution(x):
    return e ** (-sin(x)) + sin(x) - 1

def Runge_Kutta(iterations, h, x_0, y_0, f):
    x = x_0
    y = y_0
    for _ in range(iterations):
        k1 = h * f(x, y)
        k2 = h * f(x + h / 2, y + k1 / 2)
        k3 = h * f(x + h / 2, y + k2 / 2)
```

```
        k4 = h * f(x + h, y + k3)
        x += h
        y += (k1 + 2 * k2 + 2 * k3 + k4) / 6
    return x, y

def Euler(iterations, h, x_0, y_0, f):
    x = x_0
    y = y_0
    for _ in range(iterations):
        k1 = h * f(x, y)
        x += h
        y += k1
    return x, y

import matplotlib.pyplot as plt

n_values = [100, 1000, 10000, 100000, 1000000]
# Plot for x = 1
plt.subplot(1, 2, 1)
plt.title('x = 1')  # Add title for the plot
errors_rk_x1 = []
errors_euler_x1 = []
for n in n_values:
    x, y = Runge_Kutta(n, 1 / n, 0, 0, f)
    error_rk = abs(y - exact_solution(x))
    errors_rk_x1.append(error_rk)

    x, y = Euler(n, 1 / n, 0, 0, f)
    error_euler = abs(y - exact_solution(x))
    errors_euler_x1.append(error_euler)

plt.plot(n_values, errors_rk_x1, label="Runge-Kutta")
plt.plot(n_values, errors_euler_x1, label="Euler")
plt.xlabel('n')
plt.ylabel('Error')
plt.xscale('log')
plt.yscale('log')
plt.legend()

# Plot for x = 2
plt.subplot(1, 2, 2)
plt.title('x = 2')  # Add title for the plot
errors_rk_x2 = []
errors_euler_x2 = []
for n in n_values:
    x, y = Runge_Kutta(n, 2 / n, 0, 0, f)
    error_rk = abs(y - exact_solution(x))
    errors_rk_x2.append(error_rk)

    x, y = Euler(n, 2 / n, 0, 0, f)
    error_euler = abs(y - exact_solution(x))
    errors_euler_x2.append(error_euler)
```

```
plt.plot(n_values, errors_rk_x2, label="Runge-Kutta")
plt.plot(n_values, errors_euler_x2, label="Euler")
plt.xlabel('n')
plt.ylabel('Error')
plt.xscale('log')
plt.yscale('log')
plt.legend()

plt.show()
```

Wynik programu:



Rys 1: Porównanie dokładności metod Eulera i Runge-Kutty w zależności od n, przy zadanym x

Wnioski:

Metoda Rungego-Kutty jest metodą dużo dokładniejszą niż metoda Eulera, wraz ze wzrostem liczby iteracji zwiększa się precyzja metody Eulera, w metodzie Rungego-Kutty nie jest to regułą.

Zad2:

Dane jest zagadnienie brzegowe: $y'' + y = x$, $y(0) = 1$, $y(0.5\pi) = 0.5\pi - 1$
Znaleźć rozwiązanie metodą strzałów.
Porównać otrzymane rozwiązanie z rozwiązaniem dokładnym $y(x) = \cos x - \sin x + x$.

Metoda strzałów polega na zastąpieniu zagadnienia brzegowego:

$$\begin{aligned} y(x_0) &= y_0, \\ y(x_1) &= y_1, \end{aligned} \quad => \quad \begin{aligned} y_a(x_0) &= y_0, \\ y_a'(x_0) &= a, \end{aligned}$$

gdzie parametr a należy dobrać w ten sposób, aby był on miejscem zerowym funkcji

$$F(a) = y_a(x_1) - y_1.$$

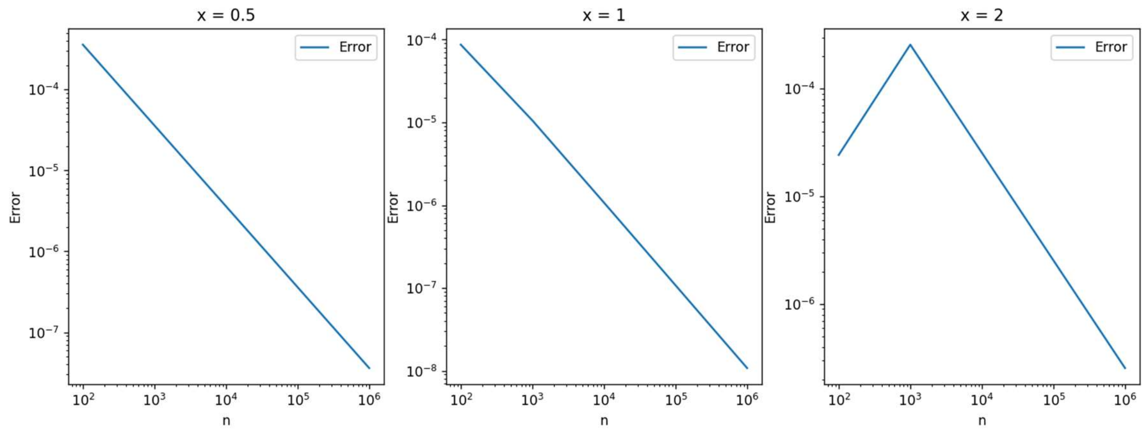
Parametru 'a' można poszukiwać np. metodą bisekcji w połączeniu z metodą Rungego - Kutty.

Użyty kod python:

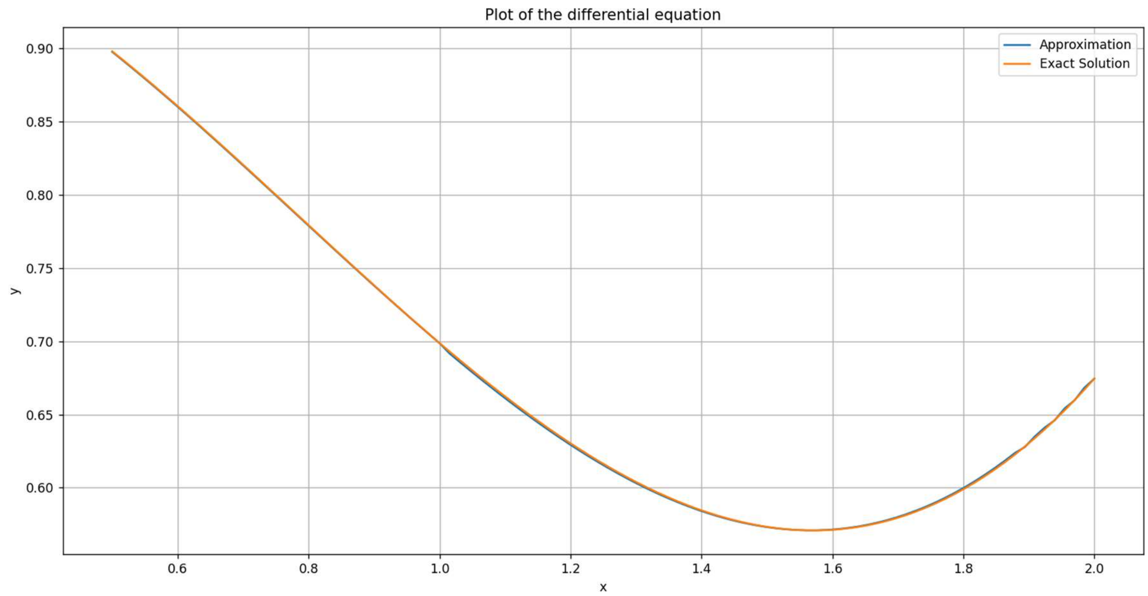
```
from math import sin, cos, pi
import numpy as np
import numpy as np

def f(x, y, y_prim):
   return x - y

def exact_solution(x):
   return cos(x) - sin(x) + x

def Runge_Kutta_for_hit(iterations, h, x_0, y_0, a, func):
   x = x_0
   y = y_0

   for _ in range(iterations):
      k1 = h * func(x, y, a)
      k2 = h * func(x + h / 2, y + k1 / 2, a)
      k3 = h * func(x + h / 2, y + k2 / 2, a)
      k4 = h * func(x + h, y + k3, a)
      delta_a = (k1 + 2 * k2 + 2 * k3 + k4) / 6

      k1 = h * a
      k2 = h * (a + h * func(x + h / 2, y + k1 / 2, a))
      k3 = h * (a + h * func(x + h / 2, y + k2 / 2, a))
      k4 = h * (a + h * func(x + h, y + k3, a))
      delta_y = (k1 + 2 * k2 + 2 * k3 + k4) / 6

      x += h
      y += delta_y
      a += delta_a

   return x, y

def hit_method(final_iterations, a_0, a_1, x_0, y_0, x_1, y_1, func, h, epsilon):
   bisect_iterations = int((x_1 - x_0) / h)
   a = (a_0 + a_1) / 2
   y = Runge_Kutta_for_hit(bisect_iterations, h, x_0, y_0, a, func)[1]
   i = 0

   while abs(y - y_1) > epsilon:
      if (y - y_1) * (Runge_Kutta_for_hit(bisect_iterations, h, x_0, y_0, a_0, func)[1] - y_1) > 0:
         a_0 = a
      else:
         a_1 = a
```

```
        a = (a_0 + a_1) / 2
        y = Runge_Kutta_for_hit(bisect_iterations, h, x_0, y_0, a, func)[1]
        i += 1

    return Runge_Kutta_for_hit(final_iterations, h, x_0, y_0, a, func)

def generate_graph():
    import matplotlib.pyplot as plt
    n = 100
    x_vals = np.linspace(0.5, 2, n)
    y_vals_hit = [hit_method(n, -100, 100, 0, 1, pi / 2, pi / 2 - 1, f, x / n, 1e-3)[1] for x in x_vals]
    y_vals_exact = [exact_solution(x) for x in x_vals]

    # Narysuj wykres
    plt.plot(x_vals, y_vals_hit, label='Approximation')
    plt.plot(x_vals, y_vals_exact, label='Exact Solution')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.title('Plot of the differential equation')
    plt.legend()
    plt.grid(True)
    plt.show()

def generate_err_graph():
    import matplotlib.pyplot as plt

    x_vals = [0.5, 1, 2]
    n_vals = [100, 1000, 10000, 100000, 1000000]
    for i, x_val in enumerate(x_vals):
        plt.subplot(1, 3, i + 1)
        plt.title(f'x = {x_val}')
        y_graph_values = []
        for n in n_vals:
            y_val = exact_solution(x_val)
            y_approx = hit_method(n, -100, 100, 0, 1, pi / 2, pi / 2 - 1, f, x_val / n, 1e-3)[1]
            y_graph_values.append(abs(y_val - y_approx))

        plt.plot(n_vals, y_graph_values, label='Error')
        plt.xlabel('n')
        plt.ylabel('Error')
        plt.xscale('log')
        plt.yscale('log')
        plt.legend()
    plt.show()

if __name__ == '__main__':
    generate_graph()
    generate_err_graph()
```

Wynik programu:



Rys2: Zależność błędu rozwiązania od ilości iteracji n, przy x = 0.5 , x = 1 , x = 2



Rys 3: Nakładanie się wykresów prawdziwego rozwiązania z aproksymowanym

Wnioski:

Dokładność metody strzałów jest niezwykle wysoka, biorąc pod uwagę skomplikowaną naturę zagadnienia brzegowego, do którego rozwiązywania jest przeznaczona. Zagadnienie brzegowe jest znacznie bardziej złożone niż zagadnienie początkowe. Pomimo tego, jesteśmy w stanie osiągnąć dokładność rzędu 10^-7 dla argumentów bliskich x0 oraz x1.

Bibliografia:

wykład dr inż. Katarzyna Rycerz
materiały podane na zajęciach
https://en.wikipedia.org/wiki/Runge%E2%80%93Kutta_methods