Metody Obliczeniowe w Nauce i Technice
Natalia Bratek, 21.05.2024
Laboratorium 10

Rozwiązywanie równań różniczkowych zwyczajnych

1. Treści zadań

1.1 Dane jest równanie różniczkowe (zagadnienie początkowe):

$$y' + y\cos x = \sin x \cos x \qquad y(0) = 0$$

Znaleźć rozwiązanie metodą Rungego-Kutty i metodą Eulera.

Porównać otrzymane rozwiązanie z rozwiązaniem dokładnym $y(x) = e^{-\sin x} + \sin x - 1$.

1.2 Dane jest zagadnienie brzegowe:

$$y'' + y = x \qquad y(0) = 1 \qquad y(0.5p) = 0.5p - 1$$

Znaleźć rozwiązanie metodą strzałów.

Porównać otrzymane rozwiązanie z rozwiązaniem dokładnym $y(x) = \cos x - \sin x + x$.

2. Rozwiązania

2.1. Zadanie pierwsze

$y' + y \cos x = \sin x \cos x$

$y(0) = 0$

Metoda Rungego:

Podstawowa idea metody Rungego-Kutty czwartego rzędu polega na ulepszeniu estymacji wartości funkcji *y* w kolejnym kroku, wykorzystując średnią ważoną kilku prób obliczeń pochodnej w danym kroku.

Iterację zaczynamy zadając x_0, y_0 oraz funkcję $f(x, y)$. W n-tym kroku iteracji, dla $n = 0, 1, 2, \ldots$, obliczamy następujące wielkości:

(1) $x_{n+1} = x_n + h,$

(2) $k_1 = h \cdot f\left(x_n, y_n\right),$

(3) $k_2 = h \cdot f\Big(x_n + \frac{h}{2}, y_n + \frac{k_1}{2}\Big),$

(4) $k_3 = h \cdot f\Big(x_n + \frac{h}{2}, y_n + \frac{k_2}{2}\Big),$

(5) $k_4 = h \cdot f\left(x_n + h, y_n + k_3\right),$

następnie liczymy

(6) $dy_n = \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$

i ostatecznie otrzymamy:

(7) $y_{n+1} = y_n + dy_n.$

Widać, że y_{n+1} zależy od wartości wcześniej obliczonej y_n oraz wielkości kroku h.

W ten sposób otrzymuje się (podobnie jak w innych iteracyjnych metodach rozwiązywania równań różniczkowych) kolejne wartości $y_n, n = 1, 2, \ldots$, które przybliżają szukaną funkcję y.

```python
import numpy as np

def exact_solution(x):
    return np.exp(-np.sin(x)) + np.sin(x) - 1

def dy_dx(x, y):
    return np.sin(x) * np.cos(x) - y * np.cos(x)

def runge_kutta_method(dy_dx, x0, y0, h, n):
    x_values = np.linspace(x0, x0 + n * h, n+1)
    y_values = np.zeros(n+1)
    y_values[0] = y0

    for i in range(n):
        k1 = dy_dx(x_values[i], y_values[i])
        k2 = dy_dx(x_values[i] + h/2, y_values[i] + h/2 * k1)
        k3 = dy_dx(x_values[i] + h/2, y_values[i] + h/2 * k2)
        k4 = dy_dx(x_values[i] + h, y_values[i] + h * k3)

        y_values[i+1] = y_values[i] + (h/6) * (k1 + 2*k2 + 2*k3 + k4)

    return x_values, y_values
```

Rys. 1 Implementacja runge_kutta_method

```python
def calculate_values(dydx, x0, y0, x_target, n_values):
    results = []
    for n in n_values:
        h = (x_target - x0) / n
        x_values = np.linspace(x0, x_target, n+1)
        y_values = np.zeros(n+1)
        y_values[0] = y0
        for i in range(n):
            y_values[i + 1] = runge_kutta_method(dydx, x_values[i], y_values[i], h)
        y_numerical = y_values[-1]
        y_exact = np.exp(-np.sin(x_target)) + np.sin(x_target) - 1
        error = np.abs(y_exact - y_numerical)
        results.append((n, y_numerical, y_exact, error))
    return results
x_target = np.pi
n_values = [10, 100, 1000, 10000, 100000]
results = calculate_values(dydx, x0, y0, x_target, n_values)
for result in results:
    print(f"n = {result[0]}, ŷ = {result[1]:.12f}, y = {result[2]:.12f}, Δy = {result[3]:.2e}")
```

Rys.2 Obliczanie wyników

n_values	$\hat{y}$	y	Δy
10	0.2725471473929363	0.27254693545348885	$2.12 * 10^{-7}$
100	0.2725469354731915	0.27254693545348885	$1.97 * 10^{-11}$
1000	0.27254693545349096	0.27254693545348885	$2.11 * 10^{-15}$
10000	0.27254693545348907	0.27254693545348885	$2.22 * 10^{-16}$
100000	0.2725469354534892	0.27254693545348885	$3.33 * 10^{-16}$

Tab. 1 Wyniki dla h = 1/n

n_values	$\hat{y}$	y	Δy
10	0.6501068782272044	0.6499642412576181	$1.40 * 10^{-4}$
100	0.6499641776509412	0.6499642412576225	$7.21 * 10^{-12}$
1000	0.6499642412504499	0.6499642412576567	$5.96 * 10^{-13}$
10000	0.6499642412582372	0.6499642412576416	$1.80 * 10^{-12}$
100000	0.6499642412580978	0.6499642412589327	$2.93 * 10^{-11}$

Tab. 2 Wyniki dla h = 5/n

Wnioski:
Analizując wyniki z obu tabel, możemy wyciągnąć kilka istotnych wniosków odnośnie zachowania metody numerycznej Rungego-Kutty

1. W obu przypadkach metoda Rungego-Kutty wykazuje dobrą zbieżność, co oznacza, że błędy Δ*y* maleją w miarę zwiększania liczby kroków *n*.
2. W obu tabelach widać, że dla bardzo dużych wartości *n*, gdzie krok *h* jest bardzo mały, wyniki są bardzo bliskie dokładnemu rozwiązaniu, co świadczy o wysokiej dokładności metody dla odpowiednio drobnych podziałów.
3. Tabela druga dotyczy rozwiązania dla x=5, co jest znacznie dalej niż *x*=1 w pierwszej tabeli. Mimo to, błędy dla dłuższych przedziałów również pozostają małe, co wskazuje na dobrą stabilność metody Rungego-Kutty nawet dla dłuższych symulacji.
4. W niektórych przypadkach, szczególnie dla bardzo dużych *n* (np. 100000 w drugiej tabeli), błąd wydaje się nieznacznie rosnąć. Może to wynikać z akumulacji błędów zaokrągleń w obliczeniach komputerowych, które stają się bardziej znaczące przy bardzo małych krokach *h* i dużej liczbie operacji.

Metoda Eulera
sposób rozwiązywania równań różniczkowych opierający się na interpretacji geometrycznej równania różniczkowego.

$$y_{n+1} = y_n + hf(x_n, y_n).$$

```python
import numpy as np

def exact_solution(x):
    return np.exp(-np.sin(x)) + np.sin(x) - 1

def dy_dx(x, y):
    return np.sin(x) * np.cos(x) - y * np.cos(x)

def euler_method(f, x_range, y0, step):
    x_values = np.arange(x_range[0], x_range[1] + step, step)
    y_values = np.zeros(len(x_values))
    y_values[0] = y0
    for i in range(1, len(x_values)):
        y_values[i] = y_values[i - 1] + step * f(x_values[i - 1], y_values[i - 1])
    return x_values, y_values
```

Rys.3 Implementacja euler_method

```python
def calculate_values(dydx, x0, y0, x_target, n_values):
    results = []
    for n in n_values:
        h = (x_target - x0) / n
        x_values = np.linspace(x0, x_target, n+1)
        y_values = np.zeros(n+1)
        y_values[0] = y0
        for i in range(n):
            y_values[i + 1] = euler_method(dydx, x_values[i], y_values[i], h)
        y_numerical = y_values[-1]
        y_exact = np.exp(-np.sin(x_target)) + np.sin(x_target) - 1
        error = np.abs(y_exact - y_numerical)
        results.append((n, y_numerical, y_exact, error))
    return results
x_target = np.pi
n_values = [10, 100, 1000, 10000, 100000]
results = calculate_values(dydx, x0, y0, x_target, n_values)
for result in results:
    print(f"n = {result[0]}, ŷ = {result[1]:.12f}, y = {result[2]:.12f}, Δy = {result[3]:.2e}")
```

Rys. 4 Obliczanie wyników dla euler_method

n_values	$\hat{y}$	y	Δy
10	0.26442725830740726	0.27254693545348885	$8.12 * 10^{-3}$
100	0.2718062028296542	0.27254693545348907	$7.41 * 10^{-4}$
1000	0.2724735390160294	0.27254693545348907	$7.34 * 10^{-5}$
10000	0.27253960254408427	0.272546935453546	$7.33 * 10^{-6}$
100000	0.272546022298938	0.2725469354528976	$7.33 * 10^{-7}$

Tab. 3 Wyniki dla h = 1/n

n_values	$\hat{y}$	y	Δy
10	1.035429161008517	0.6499642412576181	$3.9 * 10^{-1}$
100	0.7022805533207783	0.6499642412576225	$5.23 * 10^{-2}$
1000	0.6553783163012044	0.6499642412576567	$5.41 * 10^{-3}$
10000	0.650507545728617	0.6499642412576416	$5.43 * 10^{-4}$
100000	0.6499696764161406	0.6499642412589327	$5.44 * 10^{-5}$

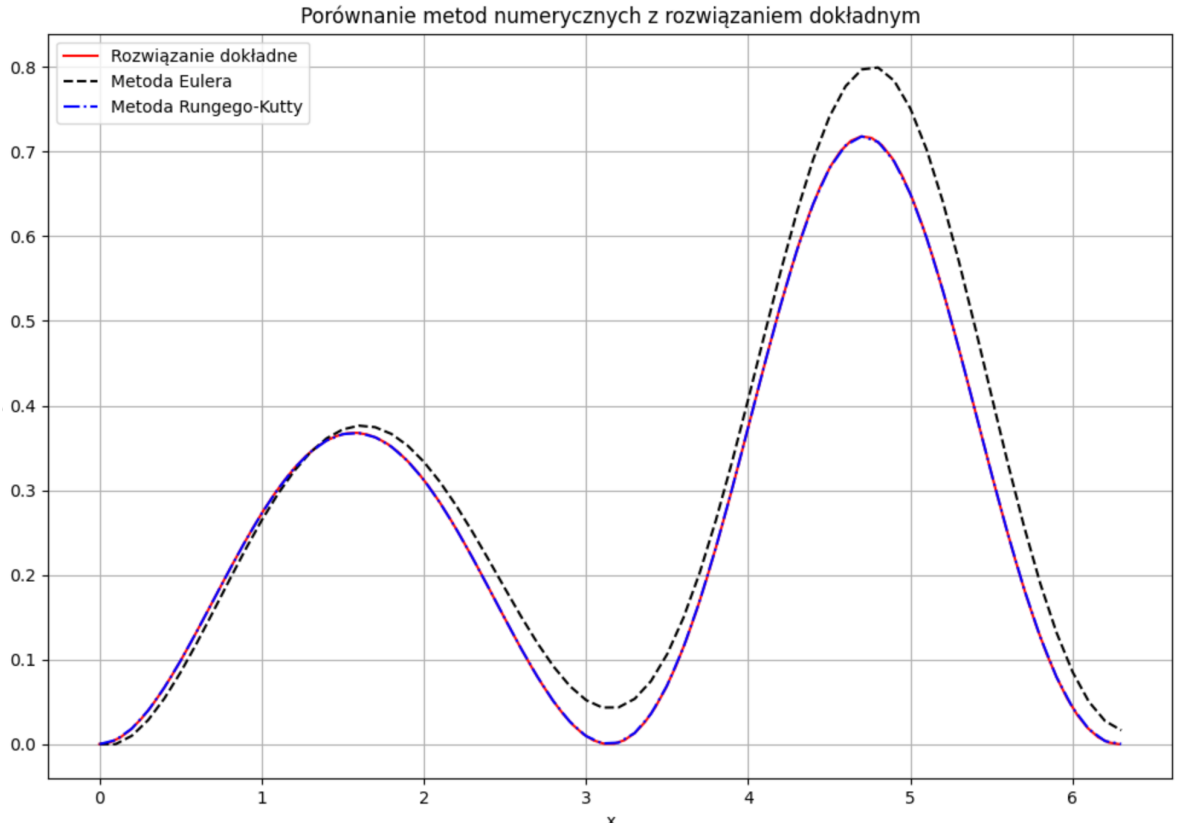
Tab. 4 Wyniki dla h = 5/n

Wnioski:

- Błąd Δy maleje w miarę zwiększania liczby kroków, co jest typową cechą metody numerycznej dla równań różniczkowych
- zmniejszanie kroku h prowadzi do znaczącego zwiększenia dokładności rozwiązania.
- większe kroki w metodzie Eulera mogą prowadzić do znacznych błędów numerycznych.

```
step_size = 0.1
x_euler, y_euler = euler_method(dydx, x_range, 0, step_size)
x_rk, y_rk = runge_kutta_method(dydx, x_range, 0, step_size)
plt.figure(figsize=(12, 8))
plt.plot(x_values, y_exact, 'k-', label='Rozwiązanie dokładne')
plt.plot(x_euler, y_euler, 'r--', label='Metoda Eulera')
plt.plot(x_rk, y_rk, 'b-.', label='Metoda Rungego-Kutty')
plt.title('Porównanie metod numerycznych z rozwiązaniem dokładnym')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid(True)
plt.show()
```

Rys. 5 Kod wykresu



Rys.6 Porównanie metod

Wnioski:

- Metoda Rungego-Kutty jest zazwyczaj znacznie dokładniejsza niż metoda Eulera dla tej samej liczby kroków. Wynika to z faktu, że Runge-Kutta wykorzystuje kilka ocen funkcji pochodnej wewnątrz każdego kroku, co pozwala lepiej aproksymować krzywą rozwiązania
- Metoda Eulera jest mniej dokładna, szczególnie dla większych kroków, co wynika z jej natury jako metody jednokrokowej, która korzysta tylko z informacji o pochodnej w punkcie początkowym każdego kroku.
- Metoda Rungego-Kutty jest bardziej stabilna numerycznie, zwłaszcza w wersjach o wyższych rzędach

- Metoda Rungego-Kutty wymaga większej liczby obliczeń na krok w porównaniu z metodą Eulera, co może prowadzić do większego obciążenia obliczeniowego, zwłaszcza w problemach wymagających bardzo małych kroków dla zachowania dokładności.

2.2. Zadanie drugie

$y'' + y = x$ $y(0) = 1$ $y(0.5\pi) = 0.5\pi - 1$

Rozwiązanie dokładne:

$y(x) = \cos x - \sin x + x$.

Metoda strzałów

Metoda strzałów to podejście, które polega na przekształceniu problemu brzegowego w jeden lub więcej problemów początkowych, które są łatwiejsze do rozwiązania numerycznie.

Proces działania metody strzałów:

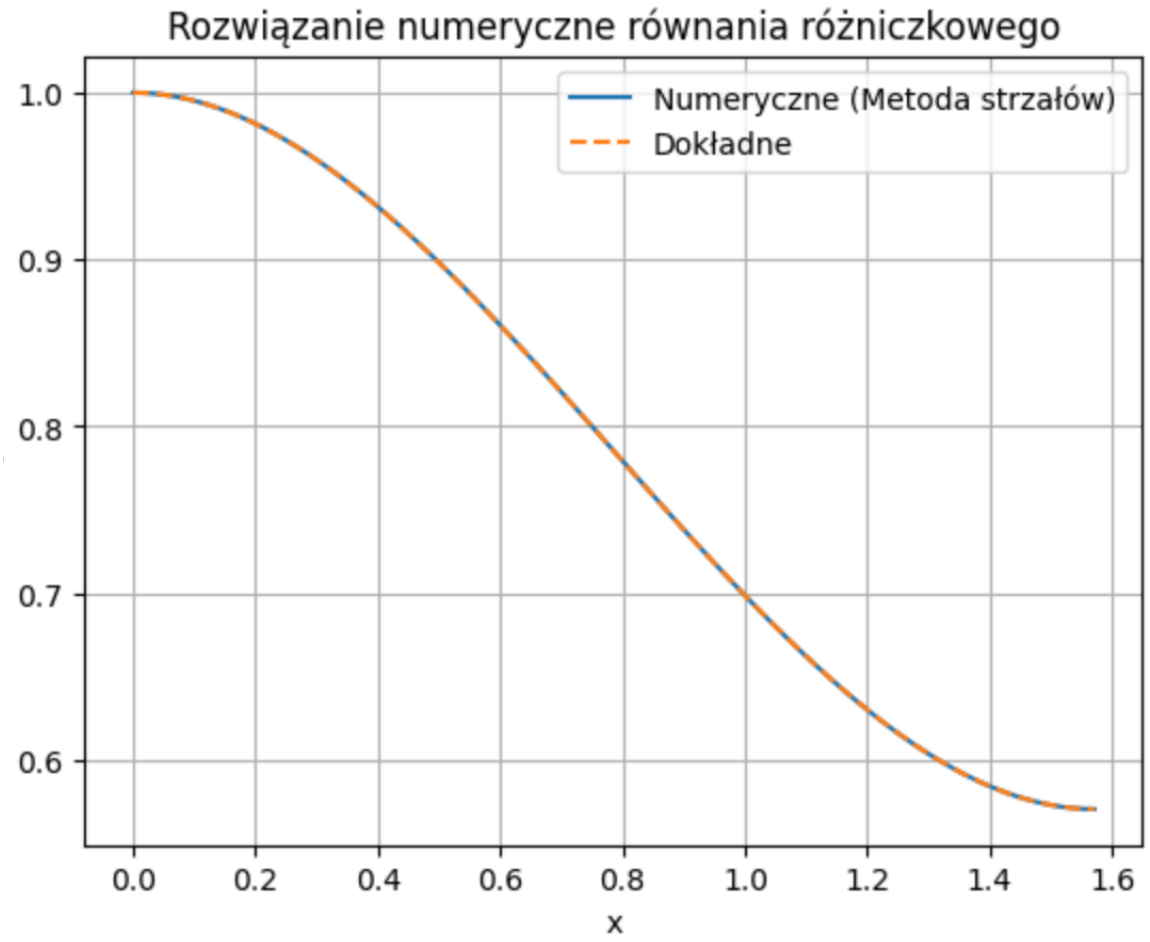
1. W metodzie strzałów, problem brzegowy, który typowo wymaga spełnienia warunków na różnych końcach przedziału (np. $y(a)$ i $y(b)$), jest przekształcany w problem wartości początkowej. Przyjmuje się założenie o wartości początkowej funkcji oraz jej pochodnej w jednym punkcie (często w punkcie a) i następnie rozwiązuje się równanie różniczkowe aż do drugiego końca przedziału.
2. Początkowo zgadujemy wartość pochodnej w punkcie początkowym (na przykład $y'(a)$). Następnie rozwiązujemy równanie różniczkowe z tym zgadnięciem jako warunkiem początkowym, poruszając się w kierunku drugiego końca przedziału. Po osiągnięciu końca przedziału sprawdzamy, czy rozwiązanie spełnia drugi warunek brzegowy (np. $y(b)$). Jeśli nie, dostosowujemy zgadnięcie i powtarzamy proces.
3. Ten proces zgadywania i sprawdzania jest powtarzany, często z wykorzystaniem metod iteracyjnych takich jak bisekcja, metoda Newtona lub metoda sekant, do momentu, gdy rozwiązanie wystarczająco dobrze spełnia warunki brzegowe.

```python
def ode_system(x, u):
    u1, u2 = u
    du1dx = u2
    du2dx = x - u1
    return [du1dx, du2dx]
y0 = 1
x_target = 0.5 * np.pi
target_y = 0.5 * np.pi - 1

def shoot_method(y_prime_guess, n):
    h = x_target / n
    sol = solve_ivp(ode_system, [0, x_target], [y0, y_prime_guess], t_eval=[x_target])
    return sol.y[0, -1]

guess_low = -10
guess_high = 10
tolerance = 1e-6
n = 1000

while abs(guess_high - guess_low) > tolerance:
    guess_mid = (guess_high + guess_low) / 2
    result_mid = shoot_method(guess_mid, n)
    if result_mid > target_y:
        guess_high = guess_mid
    else:
        guess_low = guess_mid

final_guess = (guess_high + guess_low) / 2
final_result = shoot_method(final_guess, n)
exact_solution = lambda x: np.cos(x) - np.sin(x) + x
exact_value = exact_solution(x_target)
error = np.abs(final_result - exact_value)
```

Rys.7 Metoda strzałów

```python
print(f"Ostateczne zgadnięcie dla y'(0): {final_guess}")
print(f"Numeryczna wartość y(0.5pi): {final_result}")
print(f"Dokładna wartość y(0.5pi): {exact_value}")
print(f"Błąd: {error}")
sol_final = solve_ivp(ode_system, [0, x_target], [y0, final_guess], t_eval=np.linspace(0, x_target, 100))

plt.plot(np.linspace(0, x_target, 100), sol_final.y[0], label='Numeryczne (Metoda strzałów)')
plt.plot(np.linspace(0, x_target, 100), exact_solution(np.linspace(0, x_target, 100)), label='Dokładne', linestyle='dashed')
plt.title('Rozwiązanie numeryczne równania różniczkowego')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid(True)
plt.show()
```

Rys. 8 Wyniki



Rys. 9 Wykres

```
Ostateczne zgadnięcie dla y'(0): 7.241964340209961e-05
Numeryczna wartość y(0.5pi): 0.5707962586604738
Dokładna wartość y(0.5pi): 0.5707963267948967
Błąd: 6.813442288144955e-08
```

Rys. 10 Otrzymane wyniki

Wnioski:

1. Metoda strzałów, szczególnie w połączeniu z metodami iteracyjnymi takimi jak bisekcja, okazuje się bardzo efektywna w osiąganiu dokładności w spełnianiu warunków brzegowych. Poprzez dostosowywanie wartości początkowej pochodnej *y*'(0), metoda może iteracyjnie zbliżać się do dokładnego rozwiązania, minimalizując błąd w punkcie końcowym.
2. Zwiększenie liczby kroków *n* w metodzie numerycznej zwykle prowadzi do zwiększenia dokładności rozwiązania, ponieważ mniejsze kroki pozwolą na lepsze przybliżenie dynamiki funkcji. W przypadku metody strzałów, gdzie rozwiązanie zależy od dokładnego zgadnięcia początkowych wartości, odpowiednia liczba kroków jest kluczowa do osiągnięcia precyzyjnego wyniku końcowego.

3. W przykładzie funkcji $f(x)=\cos x-\sin x+x$, gdzie forma funkcji jest dobrze znana, porównanie wskazuje, że metoda strzałów może skutecznie przybliżać rozwiązania, nawet w obecności nieliniowości, co potwierdza jej wartość w zastosowaniach praktycznych.

3. Bibliografia

- chap09-initODE.pdf
- chap10-boundODE.pdf
- https://pl.wikipedia.org/wiki/Algorytm_Rungego-Kutty
- https://pl.wikipedia.org/wiki/Metoda_Eulera
- https://pl.wikipedia.org/wiki/Metoda_strza%C5%82%C3%B3w