Laboratorium 10

Rozwiązywanie równań różniczkowych

Jan Rajczyk

24 maja 2021

1 Treści zadań

1. Dane jest równanie różniczkowe (zagadnienie początkowe):

$$\begin{cases} y' + y\cos x = \sin x \cos x \\ y(0) = 0 \end{cases}$$

Znaleźć rozwiązanie metodą Rungego-Kutty i metodą Eulera.

Porównać otrzymane rozwiązanie z rozwiązaniem dokładnym:

$$y(x) = e^{-\sin x} + \sin x - 1.$$

2. Dane jest zagadnienie brzegowe:

$$\begin{cases} y'' + y = x \\ y(0) = 1 \\ y(\frac{\pi}{2}) = \frac{\pi}{2} - 1 \end{cases}$$

Znaleźć rozwiązanie metodą strzałów.
Porównać otrzymane rozwiązanie z rozwiązaniem dokładnym:

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

2 Rozwiązania

1. Potocznie metodą **Rungego-Kutty** nazywa się metodę Rungego-Kutty 4. rzędu. Jest to metoda, w której do wyliczenia y_{n+1} potrzebujemy jedynie znać y_n. Wzór na y_{n+1} definiujemy następująco:

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

gdzie:

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

Implementację metody przedstawia poniższy kod w języku *Python*:

```
def fun(x: float, y: float):
    return sin(x) * cos(x) - y * cos(x)


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


def RungeKutta(n: int, h: float, x0: float, y0: float):
    for i in range(0, n):
        k1 = h*fun(x0, y0)
        k2 = h*fun(x0 + h/2, y0 + k1/2)
        k3 = h*fun(x0 + h/2, y0 + k2/2)
        k4 = h*fun(x0 + h, y0 + k3)
        delta = (k1 + 2*k2 + 2*k3 + k4)/6
        x1 = x0 + h
        x0 = x1
        y0 += delta
    return x0, y0
```

W celu porównania wyników tej metody oraz rozwiązania dokładnego przyjąłem $n = 10, 100, 1000, 10000, 100000, 1000000$. Przyjąłem $h = \frac{1}{n}$ i wówczas $x_n = 1$. Wyniki dla takich danych prezentują się następująco (jako $\hat{y}$ rozumiemy wartość zmierzoną, zaś y to wartość rzeczywista):

n	$\hat{y}$	y	Δy
10	0.2725471473929363	0.27254693545348885	$2.12 \cdot 10^{-7}$
100	0.2725469354731914	0.27254693545348907	$1.97 \cdot 10^{-11}$
1000	0.27254693545349107	0.27254693545348907	$2.00 \cdot 10^{-15}$
10000	0.27254693545348246	0.27254693545346	$2.25 \cdot 10^{-14}$
100000	0.27254693545351943	0.27254693545289976	$6.20 \cdot 10^{-13}$
1000000	0.2725469354528437	0.2725469354559227	$3.08 \cdot 10^{-12}$

Tablica 1: Wyniki dla $h = \frac{1}{n}$

Zauważmy, że nie zawsze wartości y są równe - dzieje się tak ze względu na to, że precyzja obliczeń przy dodawaniu bardzo małych liczb może się różnie zachowywać i stąd wynikają te minimalne różnice.

Porównałem również wyniki dla $h = \frac{5}{n}$, czyli dla $x_n = 5$. Prezentują się one następująco:

n	$\hat{y}$	y	Δy
10	0.6501068782272044	0.6499642412576181	$1.40 \cdot 10^{-4}$
100	0.6499641776509412	0.6499642412576225	$6.36 \cdot 10^{-8}$
1000	0.6499642412504499	0.6499642412576567	$7.21 \cdot 10^{-12}$
10000	0.6499642412582372	0.6499642412576416	$5.96 \cdot 10^{-13}$
100000	0.6499642412580978	0.6499642412598927	$1.80 \cdot 10^{-12}$
1000000	0.649964241276187	0.6499642413054034	$2.93 \cdot 10^{-11}$

Tablica 2: Wyniki dla $h = \frac{5}{n}$

Wnioski: Możemy zauważyć świetną precyzję metody Rungego-Kutty. Już dla 100 punktów jest to rząd 10^{-10}, co jest wynikiem naprawdę dobrym. Jedyne co może zastanawiać w tym przypadku to fakt, że wraz ze wzrostem wartości n precyzja niekoniecznie rośnie, jednakże nie trzeba zbytnio się tym martwić biorąc pod uwagę, ze są to różnice między *piko* a *femto* precyzją.

Metoda Eulera (znana też jako metoda Rungego-Kutty rzędu pierwszego) jest metodą, w której również aby obliczyć y_{n+1} potrzebujemy znać jedynie y_n. Wzór używany w tej metodzie jest zdefiniowany w sposób następujący:

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

Implementację metody przedstawia poniższy kod w języku *Python*:

```
def fun(x: float, y: float):
    return sin(x) * cos(x) - y * cos(x)


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


def Euler(n: int, h: float, x0: float, y0: float):
    for i in range(0, n):
        m = fun(x0, y0)
        y1 = y0 + h*m
        x1 = x0 + h
        x0 = x1
        y0 = y1
    return x0, y0
```

n	$\hat{y}$	y	Δy
10	0.26442725830740726	0.27254693545348885	$8.12 \cdot 10^{-3}$
100	0.2718062028296542	0.27254693545348907	$7.41 \cdot 10^{-4}$
1000	0.2724735390106294	0.27254693545348907	$7.34 \cdot 10^{-5}$
10000	0.27253960254408427	0.27254693545346	$7.33 \cdot 10^{-6}$
100000	0.2725462022298938	0.27254693545289976	$7.33 \cdot 10^{-7}$
1000000	0.2725468621311515	0.2725469354559227	$7.33 \cdot 10^{-8}$

Tablica 3: Wyniki dla $h = \frac{1}{n}$

n	$\hat{y}$	y	Δy
10	1.035429161008517	0.6499642412576181	$3.9 \cdot 10^{-1}$
100	0.7022805533207783	0.6499642412576225	$5.23 \cdot 10^{-2}$
1000	0.6553783163012044	0.6499642412576567	$5.41 \cdot 10^{-3}$
10000	0.650507545728617	0.6499642412576416	$5.43 \cdot 10^{-4}$
100000	0.6500185907513539	0.6499642412598927	$5.44 \cdot 10^{-5}$
1000000	0.6499696764161406	0.6499642413054034	$5.44 \cdot 10^{-6}$

Tablica 4: Wyniki dla $h = \frac{5}{n}$

Wnioski: Widzimy, że precyzja metody Eulera rośnie *liniowo* wraz ze wzrostem ilości rozważanych podprzedziałów. Co do jej skuteczności to jest ona naprawdę wysoka, bowiem już dla 1000 punktów, który w przypadku metod numerycznych nie jest dużą liczbą punktów precyzja metody jest na tyle dobra, że można by z niej korzystać przy większości projektów inżynierskich. Jeżeli jednak chodzi o porównanie jej z metodą Rungego-Kutty to wypada ona niekorzystnie, co jest zrozumiałe, dlatego że metoda Rungego-Kutty to po prostu nieco bardziej dopracowana metoda Eulera, która nie operuje już na jednej wartości a czterech - do tego branych z różnymi wagami.

3 Bibliografia

- `https://pl.wikipedia.org/wiki/Algorytm_Rungego-Kutty`
- `https://pl.wikipedia.org/wiki/Metoda_Eulera`
- Katarzyna Rycerz *Wykład z Metod Obliczeniowych w Nauce i Technice*
- `https://pl.wikipedia.org/wiki/Efekt_Rungego`